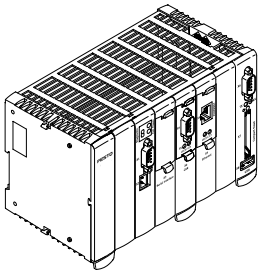
Controller

FESTO

Beschreibung

Programmier-
anleitung
CMXR
FTL-Basis



Beschreibung
560 315
de 0909a
[748985]

Ausgabe _____ 0909a

Bezeichnung _____ GDCP-CMXR-SW-DE

Bestell-Nr. _____560 315

© Festo AG & Co KG., D-73734 Esslingen, 2009

Internet: http://www.festo.com

E-Mail: service_international@festo.com

Weitergabe sowie Vervielfältigung dieses Dokuments, Verwertung und Mitteilung seines Inhalts verboten, soweit nicht ausdrücklich gestattet. Zuwiderhandlungen verpflichten zu Schadenersatz. Alle Rechte vorbehalten, insbesondere das Recht, Patent-, Gebrauchsmuster- oder Geschmacksmusteranmeldungen durchzuführen.

Verzeichnis der Revisionen

Ersteller:	
Handbuchname:	GDCP-CMXR-SW-DE
Dateiname:	
Speicherort der Datei:	

Lfd. Nr.	Beschreibung	Revisions-Index	Datum der Änderung
001	Erstellung	0805NH	25.06.2008
002	Anpassung an Version 1.20	0909a	25.08.2009

INHALTSVERZEICHNIS

1. Einleitung ... 12

2. Sicherheitshinweise ... 13

2.1 Gebrauch der Dokumentation ... 13

2.2 Bestimmungsgemäßer Gebrauch ... 13

2.3 Qualifiziertes Personal ... 14

2.4 Sicherheitshinweise zu den Produkten ... 14

2.5 Sicherheitshinweise zu diesem Handbuch ... 14

2.6 Sicherheitsanweisungen für das beschriebene Produkt ... 15

3. Programmbearbeitung ... 16

3.1 Allgemein ... 16

3.2 Interpreter ... 16

3.3 Programmstart ... 16

3.4 Satzvorlauf ... 16

3.5 Programmstruktur ... 18

3.6 FTL-Projekte ... 18

3.6.1 Globales FTL-Projekt ... 19

3.7 FTL-Programmdatei „tip“ ... 19

3.7.1 Projektglobale Programme ... 20

3.8 FTL-Datendatei „‹Name›.tid“ ... 20

3.8.1 Programmlokale Daten ... 20

3.8.2 Projektglobale Daten ... 21

3.8.3 Systemglobale Daten ... 21

3.8.4 Instanziierung von Variablen ... 22

4. Sprachaufbau ... 24

4.1 Formatierung ... 24

4.2 Bezeichner ... 24

4.3 Schlüsselwörter ... 24

4.4 Zahlenkonstante ... 25

4.5 Zeichenketten ... 25

4.6 Operatoren und Begrenzungszeichen ... 26

4.6.1 Arithmetische Operatoren ... 26

4.6.2 Logische Operatoren ... 26

4.6.3 Vergleichsoperatoren ... 26

4.6.4 Sonstige Operatoren ... 27

4.6.5 Begrenzungszeichen ... 27

5. Grunddatentypen 28
5.1 Boolescher Datentyp (BOOL) 29
5.2 Ganzzahl-Typen, REAL-Datentyp, Bitmuster-Typen 30
5.3 Klammern 30
5.4 Zeichenketten (STRING) 31
5.5 Strukturierte Datentypen 31
5.6 Arrays 31
5.6.1 Initialisierung von Arrays 34
5.6.2 Bestimmen der Arraygrenzen mit LOW und HIGH 34
5.7 Referenzvariablen (Mapping) 36
5.7.1 Deklaration von Referenzvariablen, ‹MAPTO› 37
5.7.2 Verbinden von Referenzvariablen, ‹MAP› 37
5.7.3 Prüfen von Referenzvariablen, ‹IS_MAPPED› 37
5.8 DO Anweisung 39
5.8.1 Verhalten im Überschleifbereich 41

6. Variablendeklarationen 42
6.1 Initialisierung 42

7. Ausdrücke 43
7.1 Ausführungsreihenfolge in Ausdrücken 43

8. Programmsteuerung 44
8.1 Anweisungen 44
8.2 Wertzuweisung ‹:=› 44
8.3 Bedingungen 44
8.4 Verzweigung ‹IF...THEN› 45
8.5 Sprungbefehle 46
8.5.1 Sprungmarke ‹LABEL› 47
8.5.2 Bedingter Sprung ‹IF...GOTO› 47
8.5.3 Absoluter Sprung ‹GOTO› 48
8.6 Schleifen 48
8.6.1 WHILE-Anweisung ‹WHILE› 48
8.6.2 LOOP-Anweisung ‹LOOP› 49
8.7 Unterprogramme 50
8.7.1 Unterprogrammaufruf ‹CALL› 51
8.7.2 Programmrücksprung ‹RETURN› 52
8.8 Parallele Programme 53
8.8.1 Paralleles Programm starten ‹RUN› 54
8.8.2 Paralleles Programm beenden ‹KILL› 54
8.9 Beeinflussen des Satzvorlaufes 54

8.9.1 WAIT – Anweisung mit Zeit ‹WaitTime› ... 54
8.9.2 WAIT- Anweisung mit Bedingung ‹WAIT› ... 56
8.9.3 WaitOnPath – Anweisung mit Zeit ‹WaitOnPath› ... 58
8.9.4 WaitOnPos – Anweisung mit Zeit ‹WaitOnPos› ... 60
8.10 Kommentare einfügen ‹//› ... 61
8.11 Programmzeile deaktivieren ‹##› ... 61

9. Bewegungsbefehle ... 63
9.1 Positionsbeschreibung ... 63
9.1.1 Achsposition ... 63
9.1.2 Kartesische Position ... 66
9.2 Point-To-Point-Bewegung ‹Ptp› ... 67
9.3 Relative Point-To-Point-Bewegung ‹PtpRel› ... 70
9.4 Bewegung einer Achse, ‹MoveAxisPtp›, ‹MoveAxisCart› ... 72
9.5 Linearbewegung ‹Lin› ... 74
9.6 Relative Linearbewegung ‹LinRel› ... 76
9.7 Zirkularbewegung mit Stützpunkt ... 78
9.7.1 Funktionsweise ... 78
9.7.2 Ebenendefinition ... 79
9.7.3 Zirkularbefehl mit Stützpunkt ‹CircIp› ... 81
9.7.4 Zirkularbefehl mit Stützpunkt, PTP-Anfahrt ‹PtpToCircIp› ... 83
9.7.5 Zirkularbefehl mit Stützpunkt, Lin Anfahrt ‹LinToCircIp› ... 85
9.8 Stoppen der Bewegung ‹StopMove› ... 86
9.9 Stoppen des Programms ‹StopProgram› ... 87

10. Dynamikbefehle ... 88
10.1 Autom. Begrenzung der Dynamik, Dynamiklimiter ... 89
10.2 Geschwindigkeiten ‹Vel› ... 90
10.3 Beschleunigung ‹Acc› ... 91
10.4 Ruck ‹Jerk› ... 92
10.5 Override ... 94
10.5.1 Override am Handbediengerät ‹Ovr› ... 94
10.5.2 Dynamikoverride ‹DynOvr› ... 96
10.6 Beschleunigungsrampen ... 97
10.6.1 Setzen von Rampenformen ‹Ramp› ... 98
10.7 Konstante Bahngeschwindigkeit einschalten ‹VconstOn› ... 99
10.8 Konstante Bahngeschwindigkeit ausschalten ‹VconstOff› ... 101

11. Überschleifbefehle ... 102
11.1 Nullsegmente ... 103

11.2 Grenzbereich ... 104
11.3 Geschwindigkeitsüberschleifen ... 105
11.3.1 Mit Prozentfaktor ‹OvlVel› ... 105
11.4 Geometrisches Überschleifen ... 107
11.4.1 Überschleifen der Achsen X, Y und Z ‹OvlCart› ... 107

12. Referenzsysteme (Nullpunktverschiebung) ... 109
12.1 Bezug des Referenzsystems ... 109
12.2 Daten des Referenzsystems ... 110
12.3 Referenzsystem mit direkten Werten ‹SetRefSys› ... 111
12.4 Referenzsystem mit 3 Punkten ‹SetRefSys3P› ... 112
12.5 Referenzsystem Welt ‹SetRefSysWorld› ... 114
12.6 Beispiel ... 114

13. Referenzieren einer Kinematik ... 116
13.1 Referenzfahrt ‹RefAxis› ... 116
13.2 Asynchrone Referenzfahrt ‹RefAxisAsync› ... 118
13.3 Warten auf Ende der Referenzfahrt ‹WaitRefFinished› ... 120
13.4 Status einer Achse abfragen ‹IsAxisReferenced› ... 121

14. Werkzeuge ... 122
14.1 Werkzeugdaten ... 122
14.1.1 Daten TCP-Vektor ... 122
14.2 Aktivieren von Werkzeugdaten ‹Tool› ... 125
14.2.1 Auswirkung von TCP-Daten ... 125

15. PROFIBUS-Schnittstelle ... 130
15.1 Hinweise zur Signalverarbeitung ... 131
15.2 Boolesche Ein- und Ausgänge ... 131
15.3 Integer-Variablen 32 Bit ... 132
15.4 Positionen ... 133
15.5 Referenzsysteme ... 134
15.6 Programmierter Halt ‹ProgHold› ... 135

16. Meldesystem ... 137
16.1 Meldetexte ... 137
16.2 Information ‹SetInfo› ... 139
16.3 Warnung ‹SetWarning› ... 140
16.4 Fehlermeldung ‹SetError› ... 141

17. Funktionen 142
17.1 Aktuelle Position lesen ‹ReadActualPos› 142
17.2 Zielposition lesen ‹ReadTargetPos› 143
17.3 Positionswert dauerhaft speichern ‹SavePosition› 144
17.4 Lesen der Systemzeit ‹Time› 144
17.5 Umwandeln Zeit in Text ‹TimeToStr› 145
17.6 Sinus ‹SIN›, ‹ASIN› 146
17.7 Cosinus ‹COS›, ‹ACOS› 147
17.8 Tangens ‹TAN›, ‹ATAN› 149
17.9 Cotangens ‹COT›, ‹ACOT› 150
17.10 Tangens2 ‹ATAN2› 151
17.11 Logarithmus ‹LN› 151
17.12 Exponent ‹EXP› 151
17.13 Absolutwert ‹ABS› 152
17.14 Quadratwurzel ‹SQRT› 152
17.15 Bitweise schieben ‹SHR›, ‹SHL› 153
17.16 Bitweise rotieren ‹ROR›, ‹ROL› 154
17.17 Wert nach String wandeln ‹STR› 155
17.18 ASCII Wert in Zeichen wandeln ‹CHR› 155
17.19 Zeichen in ASCII Wert wandeln ‹ORD› 155

18. Bausteine 156
18.1 Funktionen 157
18.2 Variable 157
18.3 Laufzeitverhalten 158
18.3.1 Parameter ovlEnable 158
18.4 Digitaler Eingangsbaustein DIN 159
18.4.1 Instanzierung 159
18.4.2 Methoden 161
18.4.3 Auf Zustand warten, Methode Wait / WaitN 162
18.4.4 Zustand Lesen Methode Read 162
18.4.5 Steigende Flanke Methode RisingEdge 162
18.4.6 Flanke rücksetzen Methode ResetRisingEdge 163
18.5 Digitaler Ausgangsbaustein DOUT 163
18.5.1 Instanzierung 163
18.5.2 Variable 165
18.5.3 Methoden 166
18.5.4 Auf Zustand warten, Methode Wait / WaitN 167
18.5.5 Zustand Lesen Methode Read 167

18.5.6 Steigende Flanke, Methode RisingEdge 167
18.5.7 Flanke rücksetzen, ResetRisingEdge 168
18.5.8 Setzen und rücksetzen, Methode Set / Reset 168
18.5.9 Setzen des Ausgangs, Methode Write 169
18.5.10 Setzen des Ausgangs auf bestimmte Dauer, Methode Pulse 169
18.6 Analoger Eingangsbaustein AIN 170
18.6.1 Instanzierung 170
18.6.2 Variable 172
18.6.3 Methoden 172
18.6.4 Warten bis Wert kleiner/größer, Methode WaitLss , WaitGrt 173
18.6.5 Warten bis Wert im/aus Bereich, Methode WaitIns, WaitOuts 174
18.6.6 Abfragen des Wertes, Methode Read 174
18.7 Analoger Ausgangsbaustein AOUT 175
18.7.1 Instanzierung 175
18.7.2 Variable 176
18.7.3 Methoden 177
18.7.4 Schreiben des Ausgangswertes, Methode Write 178
18.7.5 Warten bis Wert größer/kleiner, Methode WaitLss, WaitGrt 178
18.7.6 Warten bis Wert im/aus Bereich, Methode WaitIns, WaitOuts 179
18.7.7 Ausgangswert lesen, Methode Read 179
18.8 Uhrenbaustein CLOCK 180
18.8.1 Instanzierung 180
18.8.2 Methoden 181
18.8.3 Starten der Uhr, Methode Start 182
18.8.4 Stoppen der Uhr, Methode Stop 182
18.8.5 Rücksetzen der Uhr, Methode Reset 182
18.8.6 Auslesen der Uhr, Methode Read 182
18.8.7 Zeitwert in String konvertieren 183
18.9 Encoder-Baustein ENCODER 184
18.9.1 Instanzierung 184
18.9.2 Variable 185
18.9.3 Methoden 185
18.9.4 Encoder setzen, Methode Set 186
18.9.5 Encoder lesen Methode Read 187
18.10 CANopen Baustein COPDEVICE 187
18.10.1 Instanzierung 188
18.10.2 Methoden 188
18.10.3 SDO schreiben, Methode WriteSDO 189
18.10.4 SDO lesen, Methode ReadSDOSigned 189
18.10.5 SDO lesen, Methode ReadSDOUnsigned 191

19. Peripheriesignale ... 192
19.1 Arbeiten mit digitalen Ein- und Ausgängen ... 192
19.2 Arbeiten mit analogen Ein- und Ausgängen ... 192

20. Beispiele ... 194
20.1 Stoppen von Bewegungen ... 194
20.2 Arbeiten mit Ein- und Ausgangsbausteinen ... 197
20.3 Steuerung des Satzvorlaufs ... 198
20.4 Arbeiten mit Greifern ... 199
20.4.1 Vakuumgreifer ... 200
20.4.2 Pneumatische Parallelgreifer ... 203
20.4.3 Pneumatische Schwenk-Greifeinheit ... 206
20.5 Arbeiten mit der SPS-Schnittstelle ... 210
20.5.1 Aufgabenstellung ... 210
20.5.2 SPS-Schnittstelle ... 210
20.5.3 Ablaufprogramm ... 211

21. Menübaum der FTL Befehle ... 213

1. Einleitung

Dieses Dokument beschreibt den Befehlssatz FTL (**F**esto **T**each **L**anguage) der Festo CMXR Mehrachssteuerung mit dem Softwarestand ab der Version 1.20. Die Programmierung der CMXR-Steuerung erfolgt mit dem Handbediengerät CDSA-D1-VX oder mit dem PlugIn CMXR im Festo Configuration Tool (FCT).

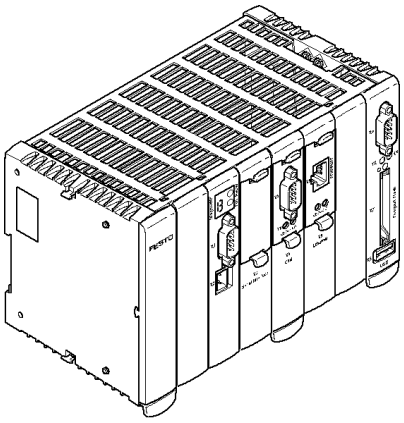
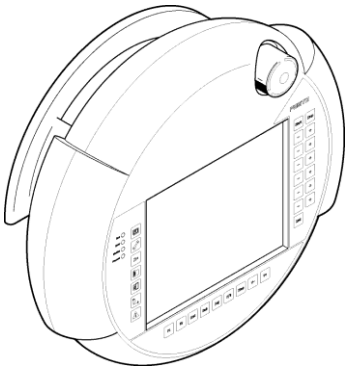


Bild 1.1 Festo Mehrachssteuerung CMXR-C1



Festo Handbediengerät CDSA-D1-VX

FTL ist eine Programmiersprache mit einem Befehlsvorrat, der für die Erstellung von Anwenderprogrammen durch Maschinen-Benutzer konzipiert ist. Der vereinfachte Sprachaufbau ermöglicht das Erstellen von einfachen Maschinenabläufen, ohne dazu eine umfangreiche Programmierausbildung zu benötigen.

Dieses Handbuch ist gültig ab der CMXR Softwareversion 1.20. Diese ist zum Softwarestand 1 kompatibel.

2. Sicherheitshinweise

2.1 Gebrauch der Dokumentation

Dieses Dokument ist für Bediener und Programmierer von Robotern gedacht, die mit dem Festo CMXR System arbeiten. Es gibt eine Einführung in die Bedienung und Programmierung. Eine entsprechende Schulung des Personals ist Voraussetzung.

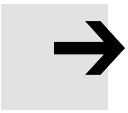
2.2 Bestimmungsgemäßer Gebrauch

Warnung

Das Festo CMXR System ist nicht für sicherheitsrelevante Steuerungsaufgaben (z.B.: Stillsetzen im Notfall oder Überwachung reduzierter Geschwindigkeiten) vorgesehen.

Das Festo CMXR System entspricht lt. EN-13849-1 nur Kategorie B und ist somit für die Realisierung von Sicherheitsfunktionen für den Personenschutz nicht ausreichend.
Für sicherheitsrelevante Steuerungsaufgaben oder Personensicherheit müssen zusätzliche externe Schutzmaßnahmen ergriffen werden, die auch im Fehlerfall einen sicheren Betriebszustand des Gesamtsystems gewährleisten.

Bei Schäden infolge von Nichtbeachtung der Warnhinweise in dieser Betriebsanleitung übernimmt Festo keine Haftung.



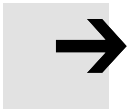
Hinweis

Vor der Inbetriebnahme sind die Sicherheitshinweise Kapitel 2.3 ff. vollständig durchzulesen.

Wenn die Dokumentation in der vorliegenden Sprache nicht einwandfrei verstanden wird, bitte beim Lieferant anfragen und diesen informieren.

Der einwandfreie und sichere Betrieb des Steuerungssystems setzt den sachgemäßen und fachgerechten Transport, die Lagerung, die Montage und die Installation sowie die sorgfältige Bedienung und die Instandhaltung voraus.

2.3 Qualifiziertes Personal



Hinweis

Für den Umgang mit elektrischen Anlagen ist ausschließlich ausgebildetes und qualifiziertes Personal einzusetzen.

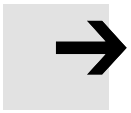
2.4 Sicherheitshinweise zu den Produkten

Warnung

GEFAHR !

Bei der Entsorgung von Batterien sind die Bestimmungen für Sondermüll zu beachten.

Obwohl Batterien eine niedrige Spannung haben, können sie doch bei Kurzschluss genug Strom abgeben, um brennbare Materialien zu entzünden. Sie dürfen deshalb nicht gemeinsam mit leitfähigen Materialien (wie z.B. Eisenspäne, mit Öl verunreinigte Drahtwolle usw.) entsorgt werden.



ESD

Elektrostatisch gefährdete Bauelemente: Unsachgemäße Handhabung kann zu Beschädigungen von Bauelementen führen.

Warnung

GEFAHR !

Gefahrbringende Bewegungen!

Lebensgefahr, schwere Körperverletzung oder Sachschaden durch unbeabsichtigte Bewegungen der Achsen!

2.5 Sicherheitshinweise zu diesem Handbuch

Warnung

GEFAHR !

Die Nichtbeachtung kann hohe Sachschäden und Personenschäden zur Folge haben.

Vorsicht

Die Nichtbeachtung kann hohe Sachschäden zur Folge haben.

2.6 Sicherheitsanweisungen für das beschriebene Produkt

Warnung

Gefahr!

Lebensgefahr durch unzureichende NOT-AUS-Einrichtungen!

NOT-AUS-Einrichtungen müssen in allen Betriebsarten der Anlage wirksam und erreichbar bleiben. Ein Entriegeln der NOT-AUS-Einrichtung darf keinen unkontrollierten Wiederanlauf der Anlage bewirken!

Erst NOT-AUS-Kette prüfen, dann einschalten!

Warnung

GEFAHR !

Gefahr für Personen und Sachen!

Testen Sie jedes neue Programm, bevor Sie die Anlage in Betrieb nehmen!

Warnung

GEFAHR !

Nachrüstungen und Veränderungen können die Sicherheit des Systems beeinträchtigen!

Die Folgen können schwere Personen-, Sach- oder Umweltschäden sein. Mögliche Nachrüstungen oder Veränderungen der Anlage mit Ausrüstungsteilen fremder Hersteller müssen daher von Festo freigegeben werden.

Warnung

GEFAHR !

Gefährliche elektrische Spannung!

Wartungsarbeiten sind, wenn nicht anders beschrieben, grundsätzlich nur bei ausgeschalteter Anlage durchzuführen! Dabei muss die Anlage gegen unbefugtes oder unbeabsichtigtes Wiedereinschalten gesichert sein.

Sind Mess- oder Prüfarbeiten an der Anlage erforderlich, müssen diese von Elektrofachkräften durchgeführt werden.

Vorsicht

Es dürfen nur von Festo zugelassene Ersatzteile verwendet werden.

3. Programmbearbeitung

3.1 Allgemein

FTL steht für **F**esto **T**each **L**anguage. FTL ist eine bewegungsorientierte Programmiersprache und dient zur Programmierung der CMXR Steuerungen.

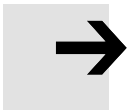
Als Programmiersprache für Robotik und Handhabungssysteme besitzt FTL einen mächtigen Sprachumfang, der jedoch leicht zu erlernen und leicht beherrschbar ist. Dies ermöglicht dem Anwender eine auf seine Applikation gerichtete Programmierung. Die Sprachbefehle basieren auf der englischen Sprache.

3.2 Interpreter

FTL-Programme werden nicht kompiliert, sondern durch einen Interpreter bearbeitet. Dieser Interpreter liest zu Beginn die Programme strukturiert in den Speicher, um sie optimal abzuarbeiten. Dieser Vorgang benötigt je nach Programmlänge eine kurze Reaktionszeit. Aus diesem Grund empfiehlt es sich vor dem Start das Programm zu laden, um einen schnellen Programmstart zu erzielen.

3.3 Programmstart

Um ein Programm zu starten, muss dieses im entsprechenden Projekt geladen werden. Das Laden kann entweder über das Handbediengerät oder durch eine externe Steuerung über eine Steuerungsschnittstelle durchgeführt werden.



Hinweis

Es kann immer nur ein Anwender-Projekt geladen sein. Das globale Systemprojekt ist zusätzlich immer geladen.

Der Programmablauf beginnt nach dem Startsignal immer an der aktuellen Position des Programmzeigers.

Hat der Programmzeiger die letzte Programmzeile des Ablaufprogramms erreicht, wird das Programm beendet, bleibt aber geladen. Sollten im Ablauf parallele Programme gestartet worden sein, so bleibt das Hauptprogramm aktiv, bis diese beendet sind.

3.4 Satzvorlauf

Um eine schnelle Programmbearbeitung zu erreichen, rechnet der Interpreter eine Anzahl von Bewegungsbefehlen voraus. Die Vorausberechnung erfolgt im laufenden Betrieb simultan zur Programmausführung und wird im System abgehandelt. Die Größe des Satzvorlaufes, d. h. die Anzahl der Bewegungsbefehle, die vorausberechnet werden, ist im System hinterlegt.

Hinweis

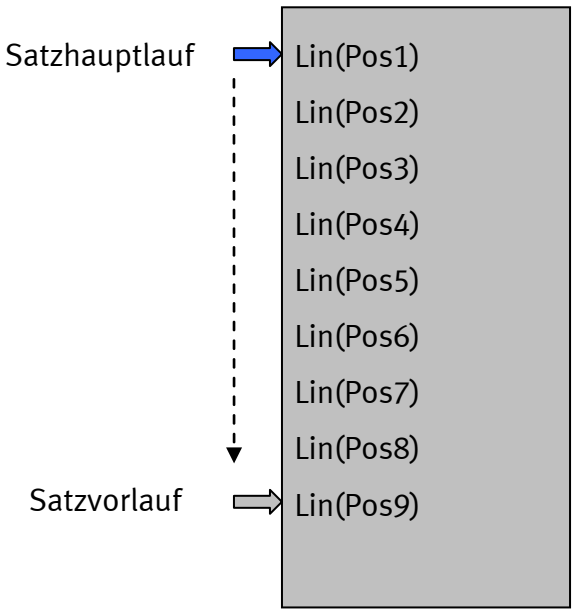
Der Satzvorlauf bezieht sich auf Bewegungsbefehle. Andere Anweisungen, die zwischen den Bewegungsbefehlen stehen, wie z. B. das Ändern von Dynamikwerten oder das Verarbeiten von Variablen, werden dadurch nicht beeinträchtigt.

In diesem Satzvorlauf werden die FTL-Befehle decodiert und für die weitere Berechnung bereitgestellt. Weiterhin erfolgt in diesem Satzvorlauf auch die Bahnplanung der Bewegung, die aufgrund der eingestellten Dynamikwerte berechnet wird.

Mit einigen Befehlen von FTL kann man den Satzvorlauf beeinflussen, d. h. er wird unter Umständen gestoppt. Dies kann z. B. bei einer Programmverzweigung der Fall sein, in der auf Eingangssignale der Peripherie entschieden wird. Hält der Satzvorlauf an, wird der gesamte im Voraus berechnete Befehlssatz abgearbeitet und die Kinematik stoppt die Bewegung. Dann wird z. B. die Entscheidung der Programmverzweigung berechnet und die folgenden Bewegungsbefehle ausgeführt.

Hinweis

Ein Stoppen des Satzvorlaufes durch einen entsprechenden Befehl hat unter Umständen ungewünschte Stopps auf der Bahn zur Folge. Ein Überschleifen der Bewegungen ist bei derartigen Befehlen nicht möglich.



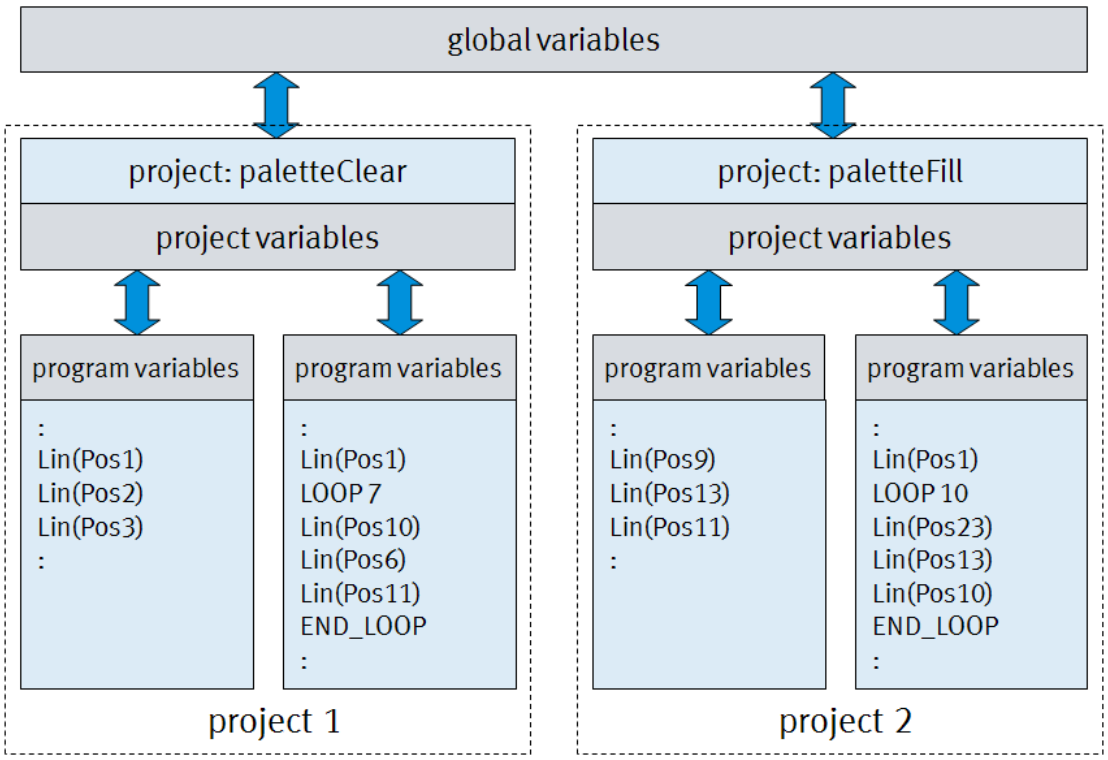
Die Abbildung zeigt einen Ausschnitt aus einem FTL-Programm. Der obere Pfeil markiert den aktuellen Befehl, der gerade ausgeführt wird. Der untere Pfeil markiert den aktiven Befehl der Satzvorausberechnung. Der Abstand der beiden Pfeile ist der Satzvorlauf.

Weitere Informationen und Beispiele zum Satzvorlauf sind im Kapitel 20.3 Steuerung des Satzvorlaufs beschrieben.

3.5 Programmstruktur

Alle FTL-Programme sind auf der Speicherkarte (Compact Flash Card) der CMXR Mehrachssteuerung im Verzeichnis "application\control\teachcontrol" (im weiteren Text Applikationsverzeichnis genannt) abgespeichert. Die Ablage der Programme erfolgt in einer Ordnerstruktur. Ein Projekt enthält die zugeordneten Bewegungsprogramme. Die Anzahl der Projekte und Programme ist durch die Größe der Speicherkarte limitiert.

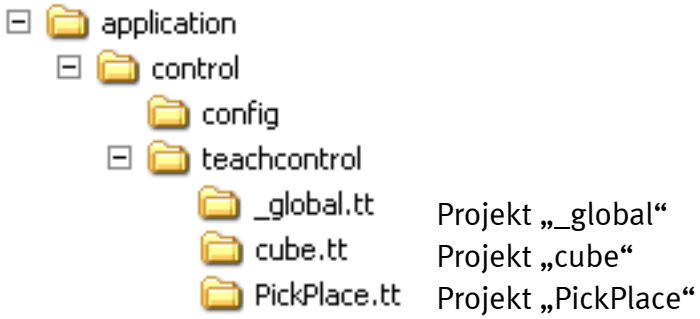
Aufbau der Programmstrukturen:



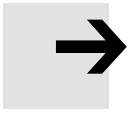
3.6 FTL-Projekte

Ein FTL-Projekt wird als Unterverzeichnis im Applikationsverzeichnis angelegt. Der Verzeichnisname ist dabei der Projektname und hat die Extension „tt“.

Beispiel für Projektnamen:



In einem Projekt werden FTL-Programme zusammengefasst. Alle Programme sind somit Elemente eines Projektes. Weitere Unterverzeichnisse in einem Projektverzeichnis sind nicht zulässig.



Hinweis

Die Projektstruktur und Verwaltung wird durch die grafische Software des Handbediengerätes und über den Editor der FCT-Software automatisch erzeugt und verwaltet.

3.6.1 Globales FTL-Projekt

Neben allen FTL-Projekten gibt es ein globales FTL-Projekt, das den Namen „_global“ trägt. Dieser Name ist fix und darf nicht verändert werden. Das Projekt ist global für alle Projekte. Alle darin gespeicherten Daten und Programme sind aus allen anderen Projekten zugänglich. Falls Daten oder Programme in mehreren Projekten benutzt werden müssen, werden diese in diesem globalen Projekt abgelegt. Somit ist die Datenkonsistenz gewahrt.

Das globale Projekt "_global" wird beim Hochlauf der CMXR automatisch geladen und aktiviert.

3.7 FTL-Programmdatei „tip“

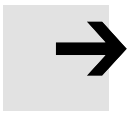
Alle zu einem Projekt gehörenden FTL-Programme werden in dem zugeordneten Projektverzeichnis abgelegt. Ein Programm setzt sich aus dem Programmcode und den dazu gehörenden Daten zusammen. Diese werden in getrennten Dateien - unter dem gleichen Namen jedoch mit einer unterschiedlichen Datei-Erweiterung (file extension) - gespeichert:

- ‹Name›.tip Datei-Erweiterung für Programmcode
- ‹Name›.tid Datei-Erweiterung für Programmdaten

Beispiel:

Für ein Programm mit dem Namen „Load“ wird die Datei "Load.tip" angelegt. Der Dateiname der programmlokalen Daten lautet "Load.tid".

Wird das Programm über das Handbediengerät CDSA oder über den Festo Programmiereditor (im Festo Configuration Tool) angelegt, erfolgt das Erzeugen der Datei für programmlokale Daten automatisch. Erfolgt eine Programmgenerierung über eine andere Software, so ist darauf zu achten, dass für jedes Programm die entsprechende Datendatei angelegt wird.



Hinweis

Erfolgt eine Programmgenerierung über eine fremde Software, so ist darauf zu achten, dass die Namenskonvention der Dateien eingehalten wird.

3.7.1 Projektglobale Programme

Das systemglobale Projekt „_global“ kann neben Daten auch Programme beinhalten. Diese Programme können von allen Programmen in allen Projekten genutzt werden.

Systemglobale Programme bieten einen entscheidenden Komfort. Programme, die von der eingesetzten Kinematik abhängen, können so losgelöst von den Applikationsprojekten definiert und abgelegt werden. Existieren bei einer Kinematik feste Positionen, wie z. B. eine Sicherheitsposition, so kann der Ablauf zum Fahren auf diese Position einmalig im globalen Projekt abgelegt werden.

3.8 FTL-Datendatei „‹Name›.tid“

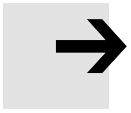
Im CMXR Steuerungssystem dienen Daten als Variable und zur Kommunikation. Die Projektstruktur bietet die Möglichkeit, Daten übersichtlich abzulegen und den Zugriff auf die Daten zu beschränken. Eine Datendatei wird durch die Datei-Erweiterung „tid“ gekennzeichnet und ist - wie die Programmdatei - textbasierend. Im System sind möglich:

- programmlokale Daten innerhalb der einzelnen Programme,
- projektglobale Daten für jedes Projekt und
- systemglobale Daten für alle Projekte.

Durch diese Aufteilung kann ausgewählt werden, wer auf die Daten Zugriff hat. Die Kommunikation zwischen Programmen bzw. Projekten lässt sich so ebenfalls steuern. Mit projektglobalen Daten kann zwischen Programmen, mit systemglobalen Daten zwischen Projekten kommuniziert werden.

3.8.1 Programmlokale Daten

Programmlokale Daten sind nur innerhalb des Programms bekannt. Andere Programme oder Projekte haben keinen Zugriff auf diese Daten.



Hinweis

Programmlokale Daten sind nur innerhalb des zugeordneten Programms bekannt und gültig. Andere Programme und Projekte haben darauf keinen Zugriff.

Die folgende Abbildung zeigt die Programme „Fill“ und „Sort“. Jedes dieser Programme besitzt lokale Daten und Programmdaten, die in den jeweiligen Dateien abgelegt sind.

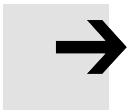
Programm **Fill**	
Datei **Fill.tid**	lokale Daten
Datei **Fill.tip**	Programm : Lin(Pos 1) Lin(Pos 2) Lin(Pos 3) Lin(Pos 4)

Programm **Sort**	
Datei **Sort.tid**	lokale Daten
Datei **Sort.tip**	Programm : Lin(Pos 10) Lin(Pos 11) Lin(Pos 12) Lin(Pos 13)

3.8.2 Projektglobale Daten

Projektglobale Daten dienen zur Kommunikation zwischen den einzelnen Programmen innerhalb eines Projektes. Programme außerhalb des entsprechenden Projektes haben keinen Zugriff auf diese Daten.

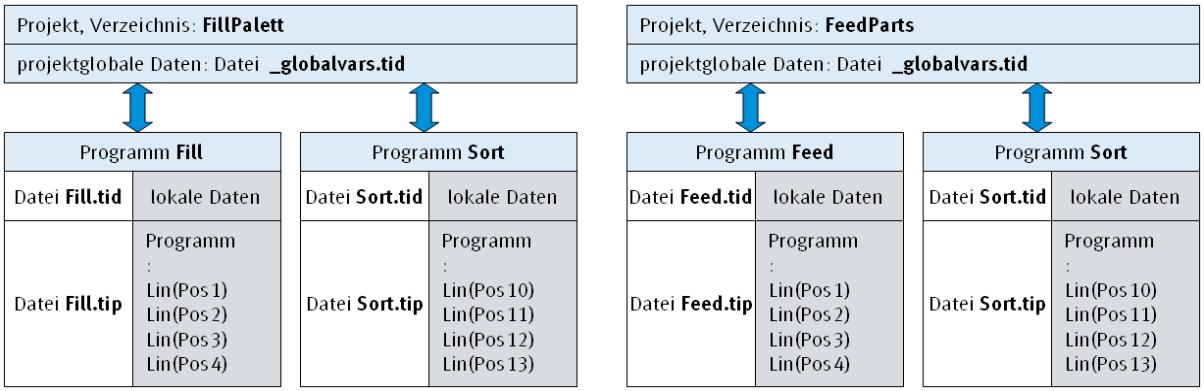
Projektglobalen Daten werden in einer eigenen Datei mit dem Namen "_globalvars.tid" gespeichert. Erfolgt eine Projektanlage über das Handbediengerät CDSA oder den FTL Programmiereditor, so wird diese projektglobale Datendatei automatisch angelegt. Bei einer Projektgenerierung über eine andere Software, z. B. kundenseitige Programmiersoftware, ist darauf zu achten, dass diese Datendatei angelegt wird.



Hinweis

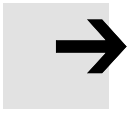
Erfolgt eine Programmgenerierung über eine fremde Software, so ist darauf zu achten, dass die Datei „_globalvars.tid“ im Projektverzeichnis angelegt wird.

Die folgende Abbildung zeigt die Projekte „FillPalett“ und „FeedParts“. Jedes dieser Projekte verfügt über projektglobale Daten, die in der Datei „_globalvars.tid“ abgespeichert und im Projektverzeichnis abgelegt sind.



3.8.3 Systemglobale Daten

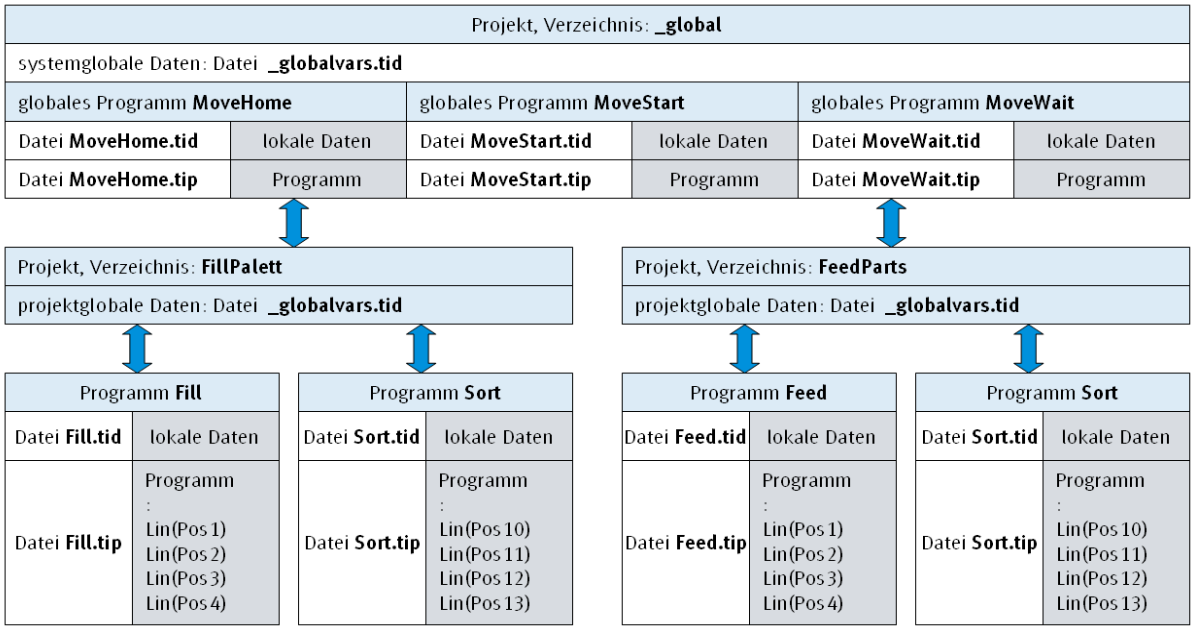
Systemglobale Daten sind systemweit in alle Programme aus allen Projekten bekannt. Diese systemglobalen Daten sind dem globalen Projekt „_global“ zugeordnet und dort in der Datendatei „_globalvars.tid“ abgelegt.



Hinweis

Da alle Programme auf systemglobale Daten Zugriff haben, ist auf die sorgfältige Verwendung dieser Daten zu achten. Ein gleichzeitiger Zugriff von mehreren Programmen auf diese Daten ist zu prüfen und ggf. in der Applikation durch entsprechende Maßnahmen zu verriegeln.

Folgende Abbildung zeigt das globale Projekt „_global“ im Verbund mit projektglobalen und lokalen Daten. Ferner sind im systemglobalen Bereich auch systemglobale Programme dargestellt.



3.8.4 Instanziierung von Variablen

Variable werden innerhalb der Datendatei „‹Name›.tid“ in folgender Form angelegt:

Syntax

‹Variablenname› **:** ‹Variablentyp› **:=** ‹Wert›

Der Name der Variablen kann frei vergeben werden. Pro Zeile kann jeweils eine Variable erstellt werden.

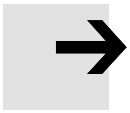
Beispiel:

```
cpos1 : CARTPOS := (100, 50, 100, 0, 0, 0)
cpos2 : CARTPOS := (600, 550, 100, 180, 0, 0)
index : DINT := 17
```

Die möglichen Variablentypen werden in den folgenden Kapiteln beschrieben.

Vorsicht

Werte von Variablen, die im Programm zur Laufzeit verändert werden, werden nicht in die Datendatei auf der Speicherkarte geschrieben. Die Werte bleiben nur solange erhalten, wie das Projekt / Programm geladen ist. Nach Abladen des Projektes / Programms oder Ausfall der Versorgungsspannung gehen im Programm veränderte Daten verloren.



Hinweis

Sollen Positionswerte dauerhaft auf der Speicherkarte gespeichert werden, so kann dies mit dem Makro „SavePosition“ geschehen.

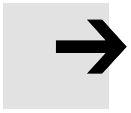
4. Sprachaufbau

4.1 Formatierung

FTL-Dateien sind lesbare Textdateien. Anweisungen bzw. Deklarationen werden durch Zeilenumbrüche getrennt. Es wird im Allgemeinen zwischen Groß- und Kleinschreibung unterschieden.

4.2 Bezeichner

Bezeichner dienen zur Identifikation von Projekten, Programmen, Bausteinen, Variablen, Konstanten und Typen. Ein Bezeichner ist eine Folge von Buchstaben, Ziffern und dem Zeichensymbol "_".



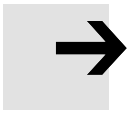
Hinweis

Zulässig sind die Zeichen a...z, A...Z, 0...9 und _ (Unterstrich). Alle anderen Zeichen sind unzulässig.

Bei Bezeichnern wird zwischen Groß- und Kleinschreibung unterschieden, d. h. es muss darauf geachtet werden, dass z. B. eine Variable in Bezug auf die Groß- und Kleinschreibung immer gleich geschrieben wird.

Beispiel:

Eine Variable mit dem Bezeichner „Index“ ist ungleich der Variablen mit dem Bezeichner „INDEX“. Dies sind 2 verschiedene Variablen.



Hinweis

Im Gegensatz zu Bezeichnern wird bei Programm- und Projektnamen nicht zwischen Groß- und Kleinschreibung unterschieden.

4.3 Schlüsselwörter

Schlüsselwörter sind Teil der FTL-Sprache. Alle Schlüsselwörter in FTL werden groß geschrieben und dürfen nicht als Namen für Programme, Variablen oder Typen verwendet werden.

Liste aller Schlüsselwörter :

CALL	IF	THEN	END_IF	ELSIF	ELSE
GOTO	LABEL	WHILE	DO	END_WHILE	LOOP
END_LOOP	RETURN	RUN	KILL	OR	XOR
MOD	AND	NOT	MAP	MAPTO	WAIT

BOOL	DINT	DWORD	REAL	STRING	ARRAY

Ebenso sind alle weiteren abgeleiteten Datentypen Schlüsselwörter.

Liste aller Funktionen :

SIN	COS	TAN	COT	LN	ABS
SQRT	EXP	ASIN	ACOS	ATAN	ATAN2
ACOT	SHR	SHL	ROR	ROL	STR
CHR	ORD	LOW	HIGH		

Zusätzlich gelten die genannten Einschränkungen für alle Befehlsnamen wie z. B. Lin, Vel, SetError usw.

4.4 Zahlenkonstante

Es wird zwischen ganzen Zahlen und reellen Zahlen unterschieden. Ganze Zahlen können in Dezimal-, Dual- oder Hexadezimaldarstellung geschrieben werden.

Beispiele für gültige ganze Zahlen:

Dezimale Schreibweise	100	-100
Duale Schreibweise	2#1010	-2#1010
Hexadezimale Schreibweise	16#1ABF	-16#1ABF

Reelle Zahlen (Gleitkommawerte) können mit Dezimalpunkt oder in Exponentialschreibweise dargestellt werden. Werte mit **Dezimalpunkt** müssen mindestens eine Nachkommastelle besitzen.

Beispiele für gültige reelle Zahlen:

Dezimale Schreibweise:	1.01	178.473	
Exponentielle Schreibweise:	1.99E4	1.99e+8	1e-8

4.5 Zeichenketten

Zeichenketten, sogenannte Strings, werden durch das Zeichen " eingeleitet und abgeschlossen. Sie dürfen alle druckbaren Zeichen beinhalten. Die Länge einer Zeichenkette ist auf 255 Zeichen beschränkt. Gültig sind alle Zeichen des ASCII Zeichensatzes.

Beispiel für eine gültige Zeichenkette:

"CMXR-C1 Mehrachssteuerung"

4.6 Operatoren und Begrenzungszeichen

Operatoren werden in Ausdrücken verwendet und beschreiben, wie Variablenwerte und Zahlenkonstanten miteinander verknüpft werden. Operatoren werden vorwiegend durch Sonderzeichen und durch Schlüsselwörter dargestellt.

4.6.1 Arithmetische Operatoren

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
MOD	Modulo Operation

Tabelle 4.1 arithmetische Operatoren

4.6.2 Logische Operatoren

Diese Operatoren können auf Wahrheitswerte und auf ganze Zahlen angewendet werden. Bei ganzen Zahlen arbeiten sie bitweise.

Operator	Bedeutung
AND	Und-Verknüpfung
OR	Oder-Verknüpfung
XOR	Exklusive Oder-Verknüpfung
NOT	Negation

Tabelle 4.2 logische Operatoren

4.6.3 Vergleichsoperatoren

Operator	Bedeutung
‹	kleiner
‹=	kleiner oder gleich
=	gleich
‹›	ungleich
›=	größer oder gleich
›	größer

Tabelle 4.3 Vergleichsoperatoren

4.6.4 Sonstige Operatoren

Operator	Bedeutung
.	Punktoperator zum Zugriff auf Strukturelemente
[ ]	Indexklammern zum Zugriff auf Arrays
()	Klammern für z. B. Parameterlisten

Tabelle 4.4 sonstige Operatoren

4.6.5 Begrenzungszeichen

Operator	Bedeutung
:=	Wertzuweisung für Variable
:	Trennzeichen bei der Instanziierung von Variablen
,	Aufzählungszeichen in Parameterlisten bei Funktions- oder Makroaufrufen

Tabelle 4.5 Begrenzungszeichen

5. Grunddatentypen

FTL unterstützt 5 Datentypen:

- Boolescher Daten
- Ganze Zahlen
- Bitmuster
- Gleitkommazahlen
- Zeichenketten

Innerhalb dieser 5 Datentypen werden in FTL elementare Datentypen abgeleitet. Aufgrund der Bedeutung des Datentyps, erfolgt eine Zuordnung für einen bestimmten Wertebereich und entsprechende Operationen.

Datentyp	Interpretation	Speichergröße	Wertebereich
BOOL	Bit	8 Bit	TRUE oder FALSE
DINT	Ganze Zahl	32 Bit	-2.147.483.648 ... +2.147.483.647
DWORD	Bitmuster	32 Bit	32 Bit
REAL	Gleitkommazahl	32 Bit	nach IEEE
STRING	Zeichenkette	max. 255 Byte	max. 255 Zeichen

Tabelle 5.1 Grunddatentypen

Je nach Datentyp sind unterschiedliche Operationen möglich.

Datentyp	Operationen
BOOL	Logische Operationen AND, OR, XOR, NOT
DINT	Arithmetikoperationen, Vergleichsoperationen
DWORD	Bitweise Operationen AND, OR, XOR, NOT, SHL, SHR, ROL, ROR, =, ‹›
REAL	Arithmetikoperationen, Vergleichsoperationen
STRING	Vergleichsoperationen, +

Tabelle 5.2 mögliche Operationen

Prinzipiell muss bei einer Zuweisung auf Datentypen der Wert vom gleichen Datentyp sein. FTL unterstützt jedoch teilweise auch die Zuweisung unterschiedlicher Datentypen. In diesem Fall wird eine automatische Typkonvertierung durchgeführt. Für bestimmte Datentypen müssen für die Zuweisung die Built-in Funktionen STR, CHR und ORD verwendet werden.

von \ nach	BOOL	Ganze Zahlen	Bitmuster	REAL	STRING
BOOL	ja	---	---	---	STR
Ganze Zahlen	---	ja	ja	ja	STR, CHR
Bitmuster	---	ja	ja	---	STR, CHR
REAL	---	ja	---	ja	STR
STRING	---	ORD	ORD	---	ja

Tabelle 5.3 mögliche Zuweisungen

Je nach Datentyp sind folgende Zuweisungen möglich:

Funktion	Operation
STR	Wandelt einen beliebigen Datentyp in einen String um.
CHR	Wandelt einen ASCII Wert in ein einzelnes Zeichen um.
ORD	Wandelt ein einzelnes Zeichnen in den ASCII Wert um.

Tabelle 5.4 Built-in Funktionen

5.1 Boolescher Datentyp (BOOL)

Der boolesche Datentyp hat die Wertigkeit TRUE und FALSE. Er wird vorwiegend für logische Operationen, auch in Verbindung mit Peripheriesignalen z. B. Sensoreingängen und Aktuator-Ausgängen verwendet.

Beispiel:

Variablen:

```
pos1          : BOOL
pos2          : BOOL
pos3          : BOOL
min1PosValid  : BOOL
allPosValid   : BOOL
```

Programmcode:

```
allPosValid  := NOT pos1 AND NOT pos2 AND NOT pos3
min1PosValid := pos1 XOR pos2 XOR pos3
```

5.2 Ganzzahl-Typen, REAL-Datentyp, Bitmuster-Typen

Die Programmiersprache FTL unterstützt ganzzahlige (auch Integer-Typen genannt), Gleitkomma- und Bitmuster-Datentypen. Je nach Fall, können diese Datentypen einander zugewiesen werden (siehe Kapitel 5 Grunddatentypen auf Seite 28). Bei solchen Zuweisungen führt das System die interne Typumwandlung automatisch durch.

Je nach Art einer Typumwandlung kann es zu einem Verlust an Genauigkeit kommen. Z. B. bei einer Wandlung von REAL nach DINT, das System entfernt die Nachkommastellen.

Beispiele für Typumwandlungen:

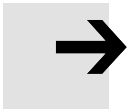
Variablen:

```
pressure : REAL
index    : DINT
```

Programmcode:

```
pressure := 1.53
index    := pressure      // Umwandlung REAL->DINT
```

Die Variable “index” enthält nach der Konvertierung den Wert “1“.



Hinweis

Typumwandlungen sollen nur dann genutzt werden, wenn die Konsequenz in jeder Hinsicht klar ist (siehe Beispiel oben).

5.3 Klammern

Mit Klammern werden Berechnungen entsprechend der Bearbeitungsreihenfolge gruppiert. Die Bearbeitung der Klammern erfolgt dabei von innen nach außen.

Beispiel:

```
:
IF (Index < 10) AND (Index > 5) THEN
:
END_IF
:
Distance := (xRow + 10) * Index
:
```

5.4 Zeichenketten (STRING)

Zeichenketten werden mit dem Datentyp STRING beschrieben. Die Länge einer Zeichenkette ist auf 255 Zeichen begrenzt. Zeichenketten, auch Strings genannt, können einander zugewiesen werden und mit Hilfe des Operators + einfach aneinander gehängt werden.

Beispiel:

Variable:

```
message : STRING
part    : STRING
```

Programmcode:

```
message := "cylinder "
part    := "DNC "
message := message + part + "is extended"
```

5.5 Strukturierte Datentypen

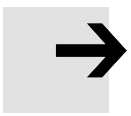
Unter strukturierten Datentypen versteht man eine feste Anordnung von Grunddatentypen zu einem neuen Datentyp.

Strukturierte Datentypen können auf der Anwenderebene nicht angelegt werden. Sie werden (nur) als Datentypen in FTL-Befehlen verwendet. Eine Anwendung ist z. B. die Darstellung von Positionen. Mit dem Datentyp "AXISPOS" wird eine Achsposition beschrieben, die aus 6 Variablen des Grunddatentyps REAL in einer bestimmter Reihenfolge besteht. Siehe Kapitel 9.1 Positionsbeschreibung auf Seite 63.

5.6 Arrays

Arrays werden verwendet um gleiche Datentypen zu einer geordneten Einheit zusammenzufassen. Der Zugriff auf einzelne Elemente des Arrays erfolgt mit einem Index. Die Grenzen des Arrays werden überwacht. Dies bedeutet, soll eine Operation außerhalb der Arraygröße ausgeführt werden, wird eine entsprechende Meldung ausgelöst.

Deklaration von Arrays:



Syntax

VAR

‹name› **: ARRAY [** ‹ Arraygröße› **] OF** ‹Datentyp› **:= (** ‹Initialisierung›**)**

END_VAR

Unter dem Parameter Arraygröße wird:

- Die Größe des Arrays,

- der Bereichsindex des Arrays angegeben,
- die Dimensionen des Arrays.

Arraygröße

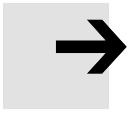
Die Arraygröße wird unter Angabe einer Konstanten (ganze Zahl) angegeben. Eine Angabe der Arraygröße mit einer Variablen ist nicht möglich.

```
// Array mit 10 Elementen
Matrix1 : ARRAY [10] OF DINT
// Array mit 12 Elementen
Matrix2 : ARRAY [12] OF CARTPOS
```

Bereichsindex

Der Bereichsindex des Arrays beginnt in der Regel bei 0 bis zu der definierten Arraygröße minus 1. Wird z. B. ein Array mit 10 Elementen definiert, geht der Arrayindex von 0 bis 9. Mit dem Arrayindex wird der Zugriff auf das einzelne Arrayelement ausgeführt.

Wenn ein spezieller Arrayindex benötigt wird, so kann dieser in die Deklaration des Arrays angegeben werden.



Syntax

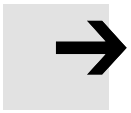
VAR
‹name› **: ARRAY [**‹start index› **...** ‹end index› **] OF**
‹Datentyp› **:= (** ‹Initialisierung›**)**
END_VAR

Folgendes Beispiel zeigt ein Array mit 10 Elementen, dessen Index von 3 bis 12 geht:

```
Matrix3 : ARRAY [3...12] OF DINT
```

Dimensionen

Ein Array ist mindestens eindimensional, kann aber auch mehrdimensional ausgeprägt sein. Mehrdimensionale Arrays werden durch Angabe der Dimensionen, getrennt durch ein Komma, innerhalb der eckigen Klammern angegeben.



Syntax
VAR
‹name› **: ARRAY [**‹size 1› **,** ‹size 2›, ‹size 3›, ... **] OF**
‹Datentyp› **:= (** ‹Initialisierung›**)**
END_VAR

Folgende Beispiele zeigen die Deklaration von mehrdimensionalen Arrays:

```
// Array mit 2 Dimensionen mit je 3 Elementen = 9 Elemente
Matrix1 : ARRAY [3, 3] OF DINT
// Array mit 3 Dimensionen mit 3 Elementen bzw. 2 Elementen =
18 Elemente
Matrix2 : ARRAY [3, 3, 2] OF DINT
```

Wenn erforderlich, kann auch bei mehrdimensionalen Arrays ein Bereichsindex angegeben werden. Folgendes Beispiel zeigt ein Array mit 3 x 3 Elementen und jeweils anderem Bereichsindex:

```
Matrix3 : ARRAY [1..3, 5..7, 0..2] OF DINT
```

5.6.1 Initialisierung von Arrays

Das System initialisiert die Arrays mit dem Wert 0. Wird eine andere Initialisierung gewünscht, so kann diese hinter dem Typ des Arrays innerhalb runden Klammern angegeben werden.

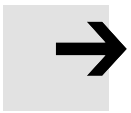
Beispiel:

```
Matrix1 : ARRAY [7] OF DINT := (0, 1, 2, 3, , , )
Matrix2 : ARRAY [3, 3] OF DINT := ((1, , 9), (3, , 7), (, , 6))
```

Jedes Feld des Arrays wird dabei mit einem Komma eingeleitet. Soll dieses Feld nicht initialisiert werden, so wird dieser nicht angegeben. Das Komma ist jedoch zwingend erforderlich, damit weitere Elemente korrekt initialisiert werden.

5.6.2 Bestimmen der Arraygrenzen mit LOW und HIGH

Mit den Schlüsselwörtern LOW kann der unterste, mit HIGH der oberste Bereichsindex eines Arrays ermittelt werden.



Syntax

‹variable : DINT› **:= LOW(** ‹Arrayvariable› **)**

‹variable : DINT› **:= HIGH(** ‹Arrayvariable› **)**

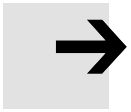
Den Funktionen LOW und HIGH wird eine Arrayvariable übergeben. Sollen die Grenzen eines eindimensionalen Arrays ermittelt werden so wird nur der Arraynamen übergeben. Bei mehrdimensionalen Arrays muss immer die Dimension angegeben werden, von der die Grenzen ausgegeben werden sollen.

```
Matrix : ARRAY [5, 19, 10..34] OF REAL
```

Die Arraybereichsgrenzen des Array sind:

- Erste Dimension 0 bis 4
    - Aufruf von LOW ergibt 0

    - Aufruf von HIGH ergibt 4
- Zweite Dimension 0 bis 18
    - Aufruf von LOW ergibt 0
    - Aufruf von HIGH ergibt 18
- Dritte Dimension 10 bis 34
    - Aufruf von LOW ergibt 10
    - Aufruf von HIGH ergibt 34



Hinweis

Wird LOW oder HIGH eine Variable übergaben, die kein Array ist, so wird eine Fehlermeldung ausgegeben.

Beispiele:

```
Matrix1    : ARRAY [1..4] OF DINT
Matrix2    : ARRAY [3,8] OF DINT
Index      : DINT

// Index hat Wert 1
Index := LOW(Matrix1)
// Index hat Wert 4
Index := HIGH(Matrix1)
// Index hat Wert 0
Index := LOW(Matrix2)
// Index hat Wert 2
Index := HIGH(Matrix2)
// Index hat Wert 0
Index := LOW(Matrix2[0])
// Index hat Wert 7
Index := HIGH(Matrix2[0])
```

5.7 Referenzvariablen (Mapping)

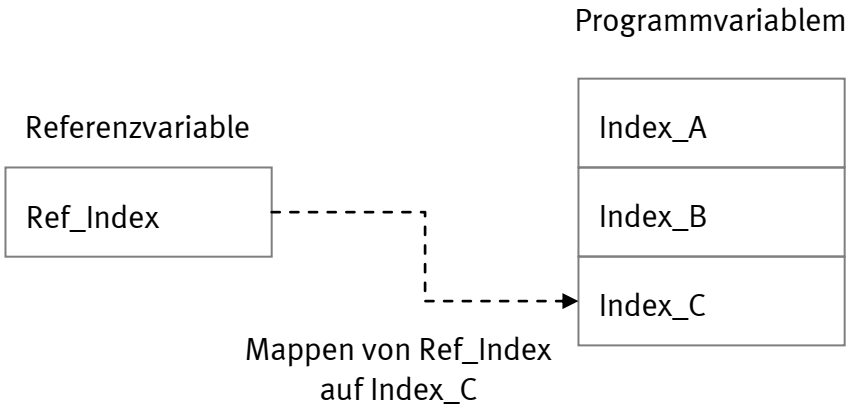
Der Mapping-Mechanismus erlaubt es, Variablen als Referenzen auf einen bestimmten Typ zu definieren, die dann mit Objekten dieses Typs verbunden werden können.

Man sagt dann, die Referenzvariable sei auf das Objekt gemappt, oder das Objekt sei der Referenzvariablen zugeordnet. Eine direkte Verwendung der Referenzvariablen bedeutet dann eigentlich eine Verwendung des zugeordneten Objekts, mit dem die Referenzvariable über den Mapping-Mechanismus verbunden ist.

Wird eine Referenzvariable direkt verwendet, obwohl es noch kein zugeordnetes Objekt gibt, so wird ein Fehler ausgelöst.

Referenzvariablen benötigen nur soviel Speicherplatz wie nötig ist, um den Verweis auf das zugeordnete Objekt abzulegen. Dieser Speicherbedarf ist unabhängig von der Art des zugeordneten Objekts.

Folgende Grafik zeigt den Mapping-Mechanismus:



Wird im Programm die Referenzvariable Ref_Index verwendet, so arbeitet diese mit dem Speicherbereich und Inhalt der Variable Index_C.

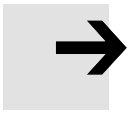
Beispiel: Die Anweisung Ref_Index := 10 beschreibt die Variable Index_C mit dem Wert 10.

Hinweis

Beim Lesen einer gemappten Variable wird die zugeordnete Variable gelesen, und beim Schreiben auf eine gemappte Variable wird auf die zugeordnete Variable geschrieben.

5.7.1 Deklaration von Referenzvariablen, ‹MAPTO›

Die Deklaration einer Referenzvariable kann mit jedem beliebigen Datentyp wie z. B. DINT oder REAL erfolgen. Die Deklaration erfolgt durch das Voranstellen des Schlüsselwortes MAPTO vor dem eigentlichen Datentyp. Eine Initialisierung der Referenzvariable ist nicht möglich.



Syntax

‹variable› : **MAPTO ‹Datentyp›**

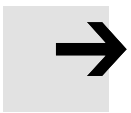
Beispiel:

```
Index : MAPTO DINT
Level : MAPTO REAL
Home  : MAPTO CARTPOS
```

In diesem Beispiel sind 3 Referenzvariablen deklariert, die unterschiedlichen Datentypen zugeordnet sind.

5.7.2 Verbinden von Referenzvariablen, ‹MAP›

Referenzvariable werden mit dem Schlüsselwort MAP auf Variable verbunden. Dabei ist zu beachten, dass nur Variablen verbunden werden können, die denselben Datentyp der Referenzvariablen besitzen.

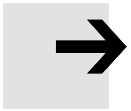


Syntax

‹variable› **: MAP(** ‹variable› **)**

5.7.3 Prüfen von Referenzvariablen, ‹IS_MAPPED›

Mit der Funktion IS_MAPPED kann überprüft werden, ob eine Referenzvariable eine Verbindung zu einer Variable besitzt. Ist eine Referenzvariable nicht verbunden und sie wird im Programm verwendet, so wird ein Fehler ausgelöst.



Syntax

IS_MAPPED (‹Referenzvariable**›) : BOOL**

Rückgabewerte:

TRUE	Referenzvariable ist verbunden
FALSE	Referenzvariable ist nicht verbunden

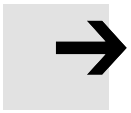
Eine Verwendung dieser Funktion ist z. B. die Verwendung von Referenzvariablen in einem Unterprogramm, wobei die Verbindung der Variablen außerhalb des Unterprogrammes erfolgt.

Beispiel:

```
IF  IS_MAPPED(Level) THEN
:
:
ELSIF
  SetError(“Not mapped”)
END_IF
```

5.8 DO Anweisung

Mit der DO Anweisung ist es möglich Anweisungen im Hauptlauf der Programmbearbeitung auszuführen. Die Anweisung besitzt folgende Syntax:

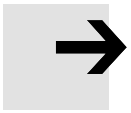


Syntax

‹FTL Makro› **DO** ‹ Makro oder Anweisung ›

Die Ausführung der Anweisung nach DO wird im Hauptlauf der Programmbearbeitung ausgeführt. Die Vorausberechnung des Programms wird durch die DO Anweisung nicht beeinflusst.

Nach einer DO Anweisung kann ein Makro oder eine Anweisung stehen. Nicht erlaubt sind Makros oder Anweisungen die sich auf eine Kinematik beziehen wie z. B. Dynamikangabe oder eine Positionsfahrt. Derartige Anweisungen sind nur vor der Do Anweisung erlaubt, da die Bahnplanung Anweisungen nach der DO Anweisung nicht berücksichtigen kann.



Hinweis

Die Ausführung der DO Anweisung erfolgt, nachdem die Anweisung davor im Hauptlauf ausgeführt wurde. Die Ausführung erfolgt bedingungslos.

Mit der DO Anweisung können im Hauptlauf der FTL Programmbearbeitung z. B. Variablen beschrieben oder Ausgänge gesetzt werden. Die DO Anweisung kann in Verbindung mit allen Bewegungsbefehlen z. B. Ptp, Lin verwendet werden.

Beispiel Setzen eines Ausgangs:

Mit dem Erreichen von pos2 wird der Ausgang des Bausteines Vacuum gesetzt.

```
:
Lin(pos1)
Lin(pos2) DO Vacuum.Set()
Lin(pos3)
:
```

Beispiel Beschreiben von Variablen an die SPS:

```
:
```

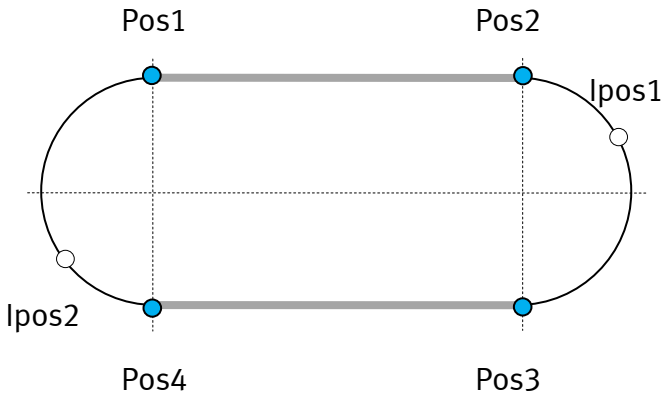
```
Lin(pos1)
Lin(pos2) DO plc_Dint[3] := 5
Lin(pos3)
:
```

Mit der DO Anweisung können einfache Schaltvorgänge ausgeführt werden, die im Hauptlauf der FTL Bearbeitung ausgeführt werden. Eine Kompensation etwaiger Schaltzeiten ist hier nicht möglich. Es erfolgt keine Kopplung an die Bahngeschwindigkeit. Dies bedeutet, dass sich durch eine Änderung der Bahndynamik das Schaltverhalten, bezogen auf die Bahn, ändern kann.

Beispiel

Auf der folgenden Kontur muss ein digitaler Ausgang auf den geraden Strecken eingeschaltet und in den Kreisbogen ausgeschaltet werden.

```
:
Lin(Pos1) DO Applicator.Set()
Lin(Pos2) DO Applicator.Reset()
CircIp(Ipos1,Pos3) DO Applicator.Set()
Lin(Pos4) DO Applicator.Reset()
CircIp(Ipos2,Pos1)
:
```

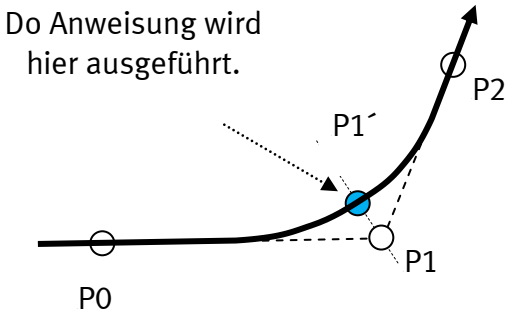


5.8.1 Verhalten im Überschleifbereich

Ist das Überschleifen aktiviert, so kann die DO Anweisung nicht beim Erreichen des Programmierten Punktes ausgeführt werden, da dieser bedingt durch das Überschleifen nicht erreicht wird. In diesem Fall wird der Schaltpunkt auf die Bahn des Überschleifbereiches projiziert und ausgeführt.

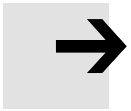
Hinweis

Münden die Bewegungssegmente tangential ineinander, so hat das Überschleifen keine Auswirkung auf das Ausführen der DO Anweisung.



6. Variablendeklarationen

Die Variablendeklarationen erfolgen in den zugehörigen Dateien mit der Datei-Endung “tid“. Darin wird der Name der Variable und der Datentyp festgelegt. Name und Datentyp werden mit einem Doppelpunkt voneinander getrennt.



Syntax

‹Name› **:** ‹Typ›

Beispiele gültiger Variablendeklarationen:

```
offset      : REAL
flag        : BOOL
index       : DINT
name        : STRING
```

6.1 Initialisierung

Alle FTL Variablen werden automatisch initialisiert. Dabei werden Integer und Real Variable mit 0, Zeichenketten (Strings) mit einer leeren Zeichenkette “ “ und boolesche Variable mit FALSE initialisiert.

Neben dieser automatischen Initialisierung gibt es auch die Möglichkeit, Variablen mit einem bestimmten Wert zu Initialisieren. Man spricht hier auch von einer expliziten Initialisierung. Der Initialwert wird dabei nach der Typangabe in Form einer Zuweisung in der Datendatei angegeben. Real Variablen können dabei auch mit Integerwerten initialisiert werden.

Beispiele gültiger Initialisierungen:

```
index       : DINT := 1
pi          : REAL := 3.1415
radius      : REAL := 10
flag        : BOOL := TRUE
message     : STRING := "Hello"
```

7. Ausdrücke

Ein Ausdruck beschreibt einen Wert, der einem bestimmten Datentyp zugeordnet ist. Ein Ausdruck kann Variablen und Funktionen enthalten. Die Bestandteile eines Ausdrucks werden durch Operatoren verknüpft.

```
wert := SQRT(a) - 2*a*b + SQRT(b)
```

7.1 Ausführungsreihenfolge in Ausdrücken

Operatoren in einem Ausdruck werden in einer bestimmten Reihenfolge bearbeitet:

1. () (Klammern)
2. [] (Array-Index)
3. NOT (Negation)
4. * / MOD AND (Multiplikation, Division, Modulo, Logisch UND)
5. + - OR XOR (Addition, Subtraktion, Logisch ODER /EXOR)
6. ‹ ‹= = ‹› ›= › (Vergleichsoperationen)

8. Programmsteuerung

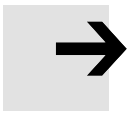
8.1 Anweisungen

FTL-Programme bestehen aus einer Aneinanderreihung von durch Zeilenumbrüche getrennten Anweisungen. FTL kennt folgende Anweisungen:

1. Wertzuweisung
2. Verzweigungen: IF, IF .. GOTO, GOTO .. LABEL, RETURN
3. Schleifen: WHILE, LOOP
4. Routinenbearbeitung: CALL, Makro-Aufruf, RUN, KILL
5. Synchronisationsanweisung: WAIT

8.2 Wertzuweisung ‹:=›

Eine Wertzuweisung besteht aus einer Variablenbezeichnung auf der linken Seite, dem Zuweisungsoperator **:=** und einem Ausdruck auf der rechten Seite. Der Datentyp des Ausdrucks muss auf den Datentyp der Variable zuweisbar sein.



Syntax

‹Variable › **:=** ‹Ausdruck›

Beispiele:

```
i := 1
x := a + b * 2
```

8.3 Bedingungen

Innerhalb von Anweisungen, Verzweigungen oder Schleifen können Bedingungen formuliert werden. Diese können boolesche Operanden (z. B. AND, OR) oder einen Vergleichsoperand (z. B. ›, ‹=,=) beinhalten. Die Verknüpfung von mehreren Vergleichen wird durch entsprechende Klammerebenen strukturiert.

Für die folgenden Beispiele werden folgende Variablen verwendet:

```
Marker : BOOL
Flag1 : BOOL
Flag2 : BOOL
Flag3 : BOOL
Index : DINT
```

Beispiele Bedingungen:

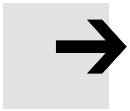
```
Marker := Index < 10
Marker := Flag1 AND Flag2 OR Flag3
IF Index < 10 THEN
:
END_IF
WHILE Index < 5 DO
:
END_WHILE
WHILE NOT Flag1 AND Flag3 OR Flag2 DO
:
END_WHILE
```

Beispiele Bedingungen mit Klammern:

```
Marker := (Index < 10) AND (Index < 0)
Marker := Flag1 AND (Index < 7)
IF (Index < 10) AND (Index > 5) THEN
:
END_IF
```

8.4 Verzweigung ‹IF...THEN›

Die IF-Anweisung ermöglicht Verzweigungen im Programmablauf, die von Bedingungen abhängen. Die Bedingung muss vom Datentyp BOOL sein und kann aus mehreren Operanden bestehen. Mit der Anweisung ELSIF lassen sich mehrere Bedingungen formulieren. Mit der Anweisung ELSE können Anweisungen definiert werden, die angesprungen werden, falls keine der Bedingungen zutrifft. Die IF-Anweisung wird mit der Anweisung END_IF abgeschlossen.



Syntax

IF ‹Bedingung› **THEN**
‹Anweisungen›
ELSIF ‹Bedingung› **THEN**
‹Anweisungen›
ELSE
‹Anweisungen›
END_IF

Die ELSEIF-Anweisung kann mehrfach vorkommen.

Beispiel:

Von einer Messanlage werden 2 Signale gesendet, die eine Qualitätsbeurteilung zulassen.

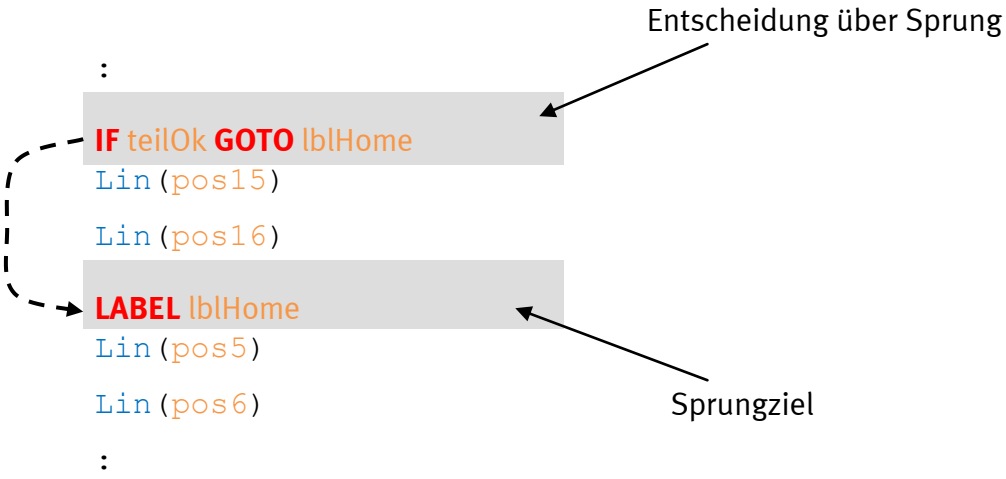
partOk	Signal für Gutteil
partBad	Signal für Ausschuss

Im folgenden FTL-Beispiel wird im Falle eines Gutteiles und eines Ausschussteiles entsprechend verzweigt. Sind die Signale undefiniert d. h. beide Signale haben den Zustand TRUE oder FALSE, so wird auf die Position PosStart gefahren.

```
IF partOk AND NOT partBad THEN     // Gutteil
  Lin(pos12)
  Lin(pos13)
ELSIF NOT partOk AND partBad THEN  // Ausschussteil
  Lin(pos3)
  Lin(pos4)
ELSE
  Lin(posStart)      // keine definierten Signale
END_IF
```

8.5 Sprungbefehle

Oft müssen in FTL-Programmen Sprünge ausgeführt werden. Diese Sprünge können auf einer Entscheidung beruhen oder bedingungslos ausgeführt werden. Um einen Sprung durchzuführen benötigt man einen Ausgangspunkt und ein Sprungziel.



Hinweis

Das Sprungziel muss sich innerhalb des aktiven Programms befinden. Ein Sprung in einen Anweisungsblock mit IF..THEN, WHILE oder LOOP ist nicht erlaubt. Jedoch darf aus einem solchen Anweisungsblock heraus gesprungen werden.

8.5.1 Sprungmarke ‹LABEL›

Eine Sprungmarke muss als Ziel eines Sprunges deklariert werden. Die Sprungmarke besteht aus einem Namen, der frei vergeben werden kann. Die Sprungmarke ist eindeutig und darf nur einmal in einem Programm vorkommen. Damit das System eine Sprungmarke erkennt, wird das Schlüsselwort LABEL vorangestellt.

Syntax

LABEL ‹Sprungmarke›

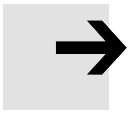
Die Sprungmarke kann mit Sprungbefehlen angesprungen werden, die im Folgenden beschrieben werden.

Hinweis

Bei der Programmierung mit dem CDSA muss zunächst das Sprungziel (LABEL) definiert werden. Erst anschließend kann die GOTO Anweisung programmiert werden.

8.5.2 Bedingter Sprung ‹IF...GOTO›

Mit der IF...GOTO Anweisung werden bedingte Sprünge ausgeführt. Diese Anweisung benötigt wie die Programmverzweigung IF...THEN eine Bedingung, die ein Ergebnis vom Datentyp BOOL liefern muss.



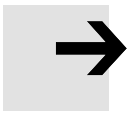
Syntax

IF ‹Bedingung› **GOTO** ‹Sprungmarke›

Ist die Bedingung erfüllt, d. h. das Ergebnis ist TRUE, dann wird der Sprung ausgeführt. Ist die Bedingung nicht erfüllt, d. h. das Ergebnis ist FALSE, so werden die Befehle in den nachfolgenden Programmzeilen ausgeführt.

8.5.3 Absoluter Sprung ‹GOTO›

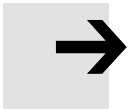
Im Gegensatz zur bedingten Sprunganweisung IF...GOTO, ist die GOTO Anweisung absolut. Dies bedeutet, es gibt keine Bedingungsanweisung.



Syntax

GOTO ‹Sprungmarke›

Mit der GOTO Anweisung lassen sich Programmteile sehr leicht überspringen. Sie eignet sich auch dazu, um aus Programmschleifen zu springen.



Hinweis

Bei der Programmierung mit dem CDSA muss zunächst das Sprungziel (LABEL) definiert werden. Erst anschließend kann die GOTO Anweisung programmiert werden.

8.6 Schleifen

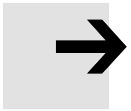
Schleifen sind Programmanweisungen, die bedingt oder unbedingt einen definierten Programmteil wiederholen. Dadurch lassen sich je nach Aufgabe Programme sehr kurz fassen, z. B. eine Entnahme von Teilen aus einer Palette, deren Positionen sich aufgrund der bekannten Anordnung berechnen lassen.

Vorsicht

Durch die Programmierung von Endlosschleifen kann das Laufzeitverhalten der CMXR Steuerung beeinträchtigt werden. Innerhalb Endlosschleifen müssen Anweisungen wie Wait oder WaitTime stehen, damit eine Endlosschleife die CMXR Steuerung nicht blockiert.

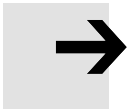
8.6.1 WHILE-Anweisung ‹WHILE›

Die WHILE-Anweisung dient zur Wiederholung einer Anweisungsfolge, solange eine Bedingung erfüllt ist. Das Ergebnis der Schleifenbedingung muss vom Datentyp BOOL sein. Die Schleife darf beliebig viele Anweisungen enthalten. Die WHILE-Anweisung wird mit dem Schlüsselwort END_WHILE abgeschlossen.



Hinweis

Wenn eine Schleife sehr oft durchlaufen wird und keine WAIT-Anweisungen enthält, kann sie die Ausführung anderer FTL-Programme behindern. Üblicherweise wird die Steuerung so konfiguriert sein, dass in diesem Fall eine Warnung ausgelöst und das unkooperative Programm kurzfristig ausgesetzt wird.



Syntax

WHILE ‹Bedingung› **DO**

‹Anweisungen›

END_WHILE

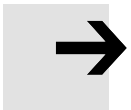
Beispiel :

```
WHILE index < 5 DO
  Lin(pos1)
  Lin(pos2)
  Lin(pos3)
  index := index + 1
END_WHILE
```

8.6.2 LOOP-Anweisung ‹LOOP›

Die LOOP-Anweisung dient zur Wiederholung einer Anweisungsfolge. Man kann direkt angeben, wie oft die Anweisungen durchlaufen werden sollen. Für jede LOOP-Anweisung wird automatisch eine interne Schleifenvariable angelegt, die beim Auflaufen auf die Schleife auf 1 initialisiert wird. Die Schleife wird solange durchlaufen, bis der Wert der internen Schleifenvariablen den Endwert überschreitet. Ist der Endwert kleiner als 1 wird die Schleife somit gar nicht durchlaufen, sondern übersprungen. Nach jedem Durchlauf wird der Wert der Schleifenvariable um eins erhöht und der Endwert neu berechnet.

Die Schleife darf beliebig viele Anweisungen enthalten.



Syntax

LOOP ‹Anzahl› **DO**

‹Anweisungen›

END_LOOP

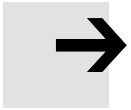
Beispiele:

```
LOOP 10 DO                  // 10 Durchläufe
  index := index + 13
END_LOOP
```

```
j := 0
...
LOOP j DO                    // wird übersprungen, da j = 0
  index := index + 13
END_LOOP
:
```

8.7 Unterprogramme

In einem Programm können andere Programme aufgerufen werden. Diese Programme müssen sich im gleichen Projekt wie das aktive Programm befinden oder im globalen Projekt (_global, siehe Kapitel 3.6.1 Globales FTL-Projekt auf Seite 19) liegen. Diese aufgerufenen Programme werden als Unterprogramme bezeichnet.



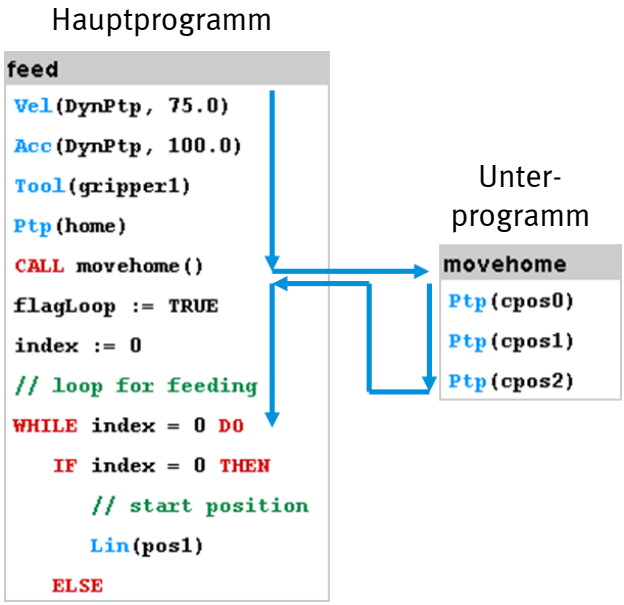
Hinweis

Ein Unterprogramm benötigt keine spezielle Kennung. Bei der Erstellung wird dieses genau wie jedes andere Programm erzeugt und in einem Projekt abgespeichert. Die Klassifizierung eines Programms als Unterprogramm erfolgt lediglich durch den Aufruf aus einem Programm und nicht direkt aus dem Projekt.

Abarbeitung:

Während das Unterprogramm abgearbeitet wird, wartet das aufrufende Programm auf dessen Ende. Nach Ende des Unterprogramms erfolgt ein automatischer Rücksprung zum aufrufenden Programm, das fortgesetzt wird.

Ablaufschema:



In diesem Beispiel ruft das Programm „feed“ ein Unterprogramm „movehome“ auf. Wenn das Programme „movehome“ die Abarbeitung beendet hat, wird das Programm „feed“ fortgesetzt.

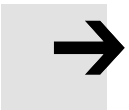
Vorteile:

Die Programmgestaltung mit Unterprogrammen ermöglicht eine überschaubare und effiziente Programmierung. Der Programmcode für Teilaufgaben wird separat abgelegt und kann von mehreren Programmen genutzt werden. Die Pflege und die Konsistenz der Programme wird dadurch gesteigert.

8.7.1 Unterprogrammaufruf ‹CALL›

Ein Unterprogrammaufruf erfolgt über den Befehl CALL. Eine Parameterübergabe an das Unterprogramm wird nicht unterstützt. Falls eine Datenübergabe notwendig ist, muss dies über globale Variable erfolgen. Diese Variablen werden vor dem Aufruf mit entsprechenden Werten versorgt und innerhalb des Unterprogramms verarbeitet.

Rekursive Aufrufe von Programmen sind nicht erlaubt. Ein Programm darf sich daher nicht selbst aufrufen. Ferner ist ein Aufruf des aufrufenden Programms nicht möglich.



Syntax

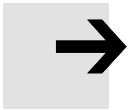
CALL ‹Programmname› ()

Beispiel:

```
:
variant := 17                // Variable für Programm Kleben
directionRight := TRUE       // Variable für Programm Kleben
directionLeft := FALSE       // Variable für Programm Kleben
CALL Glueing()               // Unterprogramm Kleben
CALL MoveStart()             // Unterprogramm fahre Startposition
:
```

8.7.2 Programmrücksprung ‹RETURN›

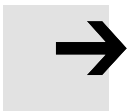
Prinzipiell wird ein Unterprogramm mit der letzten Anweisung beendet. Soll ein Unterprogramm vor der letzten Anweisung beendet werden, erfolgt dies mit dem Befehl RETURN. Das Unterprogramm wird dabei vorzeitig beendet und es wird in das aufrufende Programm zurückgesprungen, das fortgesetzt wird.



Syntax

RETURN

Eine Rückgabe von Werten an das übergeordnete Programm über den Befehl RETURN ist nicht möglich. Sollte das nötig sein, können für diese Informationsübermittlung entsprechende Variablen benützt werden.



Hinweis

Wird der Befehl RETURN im Hauptprogramm ausgeführt, wird dieses gestoppt und beendet.

Bei einem Aufruf von RETURN in einem parallelen Programm oder einem Unterprogramm wird dieses beendet. Das Hauptprogramm wird fortgesetzt.

Beispiel:

```
:
CALL CheckPart()     // Aufruf Unterprogramm CheckPart
IF NOT partOk THEN
  partStatus := 10   // globale Variable für Rückgabewert
  RETURN             // vorzeitiges Programmende
END_IF
Lin(pos3)
:
```

8.8 Parallele Programme

Ein Programm aus dem aktiven Projekt oder aus dem globalen Projekt kann auch als paralleler Prozess gestartet werden. Das Betriebssystem des CMXR sorgt dafür, dass diese Programme parallel abgearbeitet werden. Dies wird durch das interne Multitaskingsystem sichergestellt.

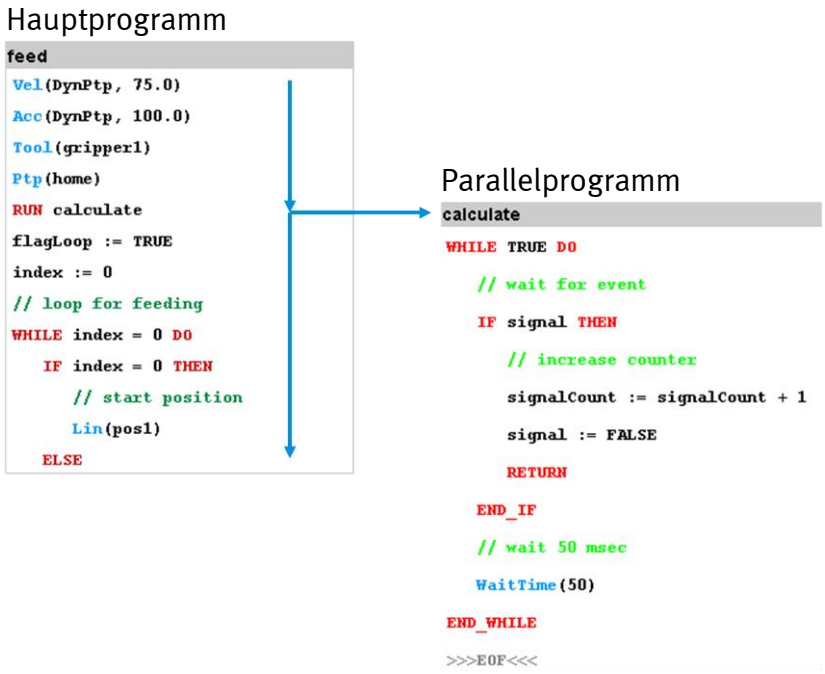
Hinweis

Fahrbefehle an eine Kinematik sind aus einem parallelen Programm nur möglich, wenn zuvor aus einem Hauptprogramm kein Fahrbefehl abgesetzt wurde. Erfolgt ein Fahrbefehl aus einem Haupt- und Parallelprogramm, erzeugt das System einen Fehler.

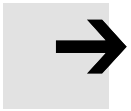
Verwendung:

Parallele Programme ermöglichen das Erstellen von Prozessen, die asynchron oder nur teilweise zum Hauptprogramm synchronisiert arbeiten. Ein Beispiel ist das Ansteuern einer Zuführ- oder Auswerfeinheit, die mit Ein-/ Ausgängen angesteuert wird. Mit logischen Befehlen und Verarbeitung der Ein-/Ausgänge, kann in diesem Fall der Ablauf beschrieben werden. Je nach Anforderung kann eine Synchronisierung zum Hauptprogramm über globale Variable erfolgen.

Ablaufschema:



In diesem Beispiel ruft das Programm „feed“ ein Parallelprogramm „calculate“ auf. Das aufgerufenen Programm „calculate“ wartet zunächst auf ein Signal und erhöht einen Zähler. Währenddessen wird das Hauptprogramm „feed“ weiter abgearbeitet.

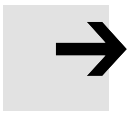


Hinweis

Wird in einem parallelen Programm eine Endlosschleife programmiert, so ist dafür zu sorgen, dass diese die Programmausführung der anderen Programme nicht blockiert. Durch einen Wait oder einen WaitTime Befehl wird ein Verteilen der Rechenleistung auf die anderen Programme gewährleistet.

8.8.1 Paralleles Programm starten ‹RUN›

Ein paralleles Programm wird mit dem Befehl RUN gestartet. Eine Parameterübergabe an das zu startende Programm ist nicht möglich. Sollten Daten übergeben werden, so können globale Variable dafür verwendet werden.



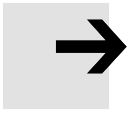
Syntax

RUN ‹Programmname›

Ein bereits laufendes Programm kann nicht gleichzeitig als Parallelprogramm gestartet werden.

8.8.2 Paralleles Programm beenden ‹KILL›

Ein aktives parallel laufendes Programm kann aus dem aufrufenden Programm über die Anweisung "KILL" beendet werden. Dabei wird zuerst das Programm und die Kinematik gestoppt, danach wird das Programm beendet.



Syntax

KILL ‹Programmname›

Auch der Befehl RETURN hat im parallelen Programm zur Folge, dass das Programm beendet wird.

8.9 Beeinflussen des Satzvorlaufes

8.9.1 WAIT – Anweisung mit Zeit ‹WaitTime›

Die WAIT Anweisung mit Zeit-Angabe ermöglicht das Programmieren einer Wartezeit. Diese Wartezeit beeinflusst das Bewegungsverhalten. Sie erzwingt ein Stoppen der Satzvorausberechnung, was zu einem Stopp der Bewegung führt. Nachdem der vorherige Befehl ausgeführt wurde, beginnt die Wartezeit zu laufen. Nach Ablauf dieser Zeit wird das Programm fortgesetzt.

Syntax

WaitTime (‹timeMS ›: DINT)

Die Angabe der Zeit erfolgt in Millisekunden und kann über einen Wert oder eine Variable angegeben werden.

Hinweis

Soll der Satzvorlauf mit dem Satzhauptlauf synchronisiert werden, so kann dies über die Anweisung WaitTime 0 erfolgen.

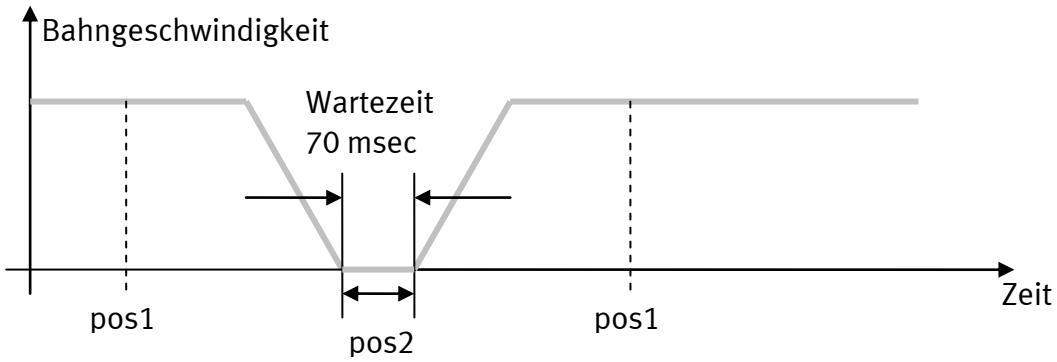
Beispiel:

Ein Handhabungssystem, bestückt mit einem Greifer, entnimmt Teile aus einer Palette. Um die Teile sicher zu greifen, muss eine gewisse Zeit abgewartet werden.

Auszug aus dem Bewegungsprogramm:

```
:
Lin(pos1)                    // fahre über Greifposition
Lin(pos2)                    // fahre auf Greiferposition
Gripper.Set()                // Greifer schließen
WaitTime(70)                 // warte 70 msec Greifzeit
Lin(pos1)                    // fahre über Greifposition
:
```

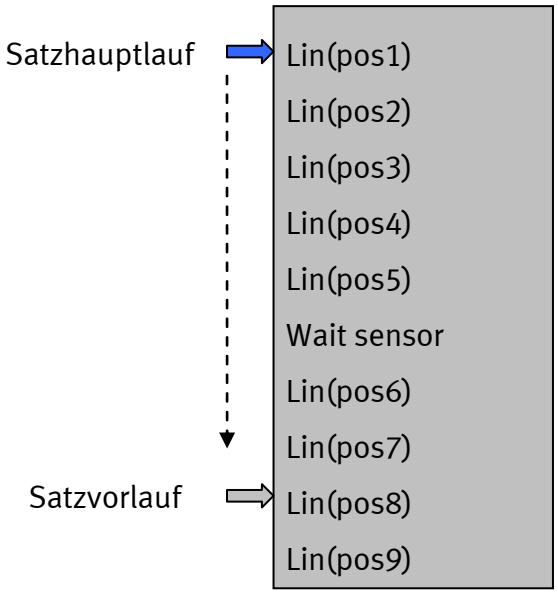
Bewegungsablauf :



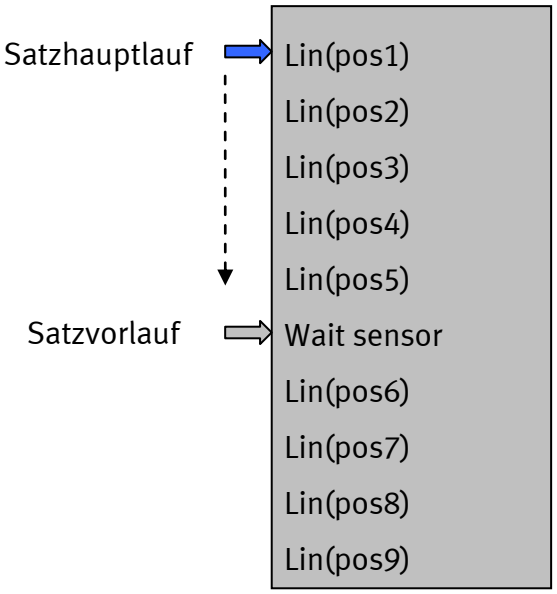
8.9.2 WAIT- Anweisung mit Bedingung ‹WAIT›

Die WAIT-Anweisung mit Bedingung ermöglicht das Abfragen von binären Zuständen. Dieser Zustand kann in Form einer einzelnen booleschen Variable, einer Kombination aus mehreren oder einer Abfrage bestehen.

Die Bearbeitung der WAIT-Anweisung erfolgt im Satzvorlauf, d. h. in der Vorausberechnung der CMXR-Steuerung. Ist die Bedingung nicht erfüllt, so wird der Satzvorlauf (Vorausberechnung) solange gestoppt, bis die Bedingung der WAIT-Anweisung erfüllt ist.



Bedingung für WAIT erfüllt, Satzvorlauf wird weiter berechnet.



Bedingung für WAIT nicht erfüllt, Satzvorlauf bleibt stehen, bis Bedingung erfüllt ist.

Ist die zeitliche Verzögerung so groß, dass sogar der Hauptlauf des Bewegungsprogramms die WAIT-Anweisung erreicht, so wird auch dieser gestoppt, was zu einem Stillstand der Bewegung führt. Die Abarbeitung und Vorausberechnung wird erst fortgesetzt, wenn die Bedingung der WAIT-Anweisung erfüllt ist.

Hinweis

Da die Abfrage der Bedingung im Satzvorlauf erfolgt, wird eine Änderung, die nach der Bearbeitung durch den Satzvorlauf erfolgt, nicht mehr registriert. Wird dies gewünscht, so kann mit dem Voranstellen des Befehls WaitTime (siehe Kapitel 8.9.1 WAIT – Anweisung mit Zeit auf Seite 54) eine Synchronisation mit dem Satzhauptlauf erzwungen werden.

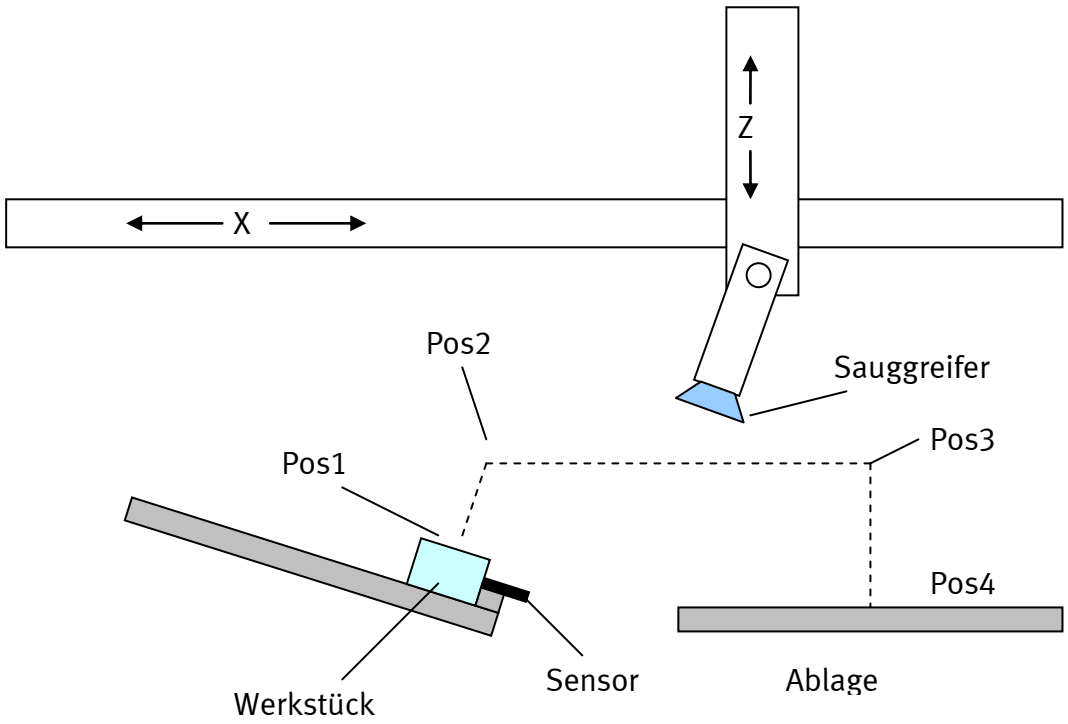
In der Bedingung der WAIT Anweisung dürfen die Grunddatentypen BOOL, REAL und DINT mit logischen Operatoren und Vergleichsoperatoren verwendet werden. Arithmetische und Bitoperatoren dürfen nicht verwendet werden.

Syntax

WAIT ‹Bedingung›

Beispiel:

Mit einer schiefen Ebene werden Teile einem Handhabungssystem zugeführt. Ein anwesendes Teil wird über einen digitalen Sensor erkannt und kann dann vom Handhabungssystem entnommen werden.



Der Sensor erkennt, ob sich ein Werkstück auf der Ablage befindet. Steht dieses bereit, wird es vom Handhabungssystem entnommen.

Auszug aus dem Bewegungsprogramm:

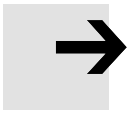
```
:
Lin(pos2)          // fahre über Teil
Vacuum.Set()       // Vakuum ein
WAIT sensor        // warte, bis Teil vorhanden
Lin(pos1)          // fahre auf Teil
Lin(pos2)          // fahre über Teil
Lin(pos3)          // fahre über Ablage
Lin(pos4)          // lege Teil ab
WaitTime(0)        // warte auf Satzhauptlauf
Vacuum.Reset()     // Vakuum aus
```

```
Lin(pos3)          // fahre über Ablage
:
```

Ist kein Werkstück in der Ablage, so wartet das Handhabungssystem über dem Werkstück, die Bewegung ist gestoppt. Meldet der Sensor „Werkstück vorhanden“, so wird das Programm fortgesetzt. Ist beim Anfahren bereits ein Werkstück vorhanden, so fährt das Handhabungssystem sofort auf das Werkstück, ohne dass die Bewegung unterbrochen wird.

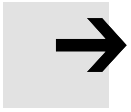
8.9.3 WaitOnPath – Anweisung mit Zeit ‹WaitOnPath›

Das Makro WaitOnPath erzeugt eine Wartezeit, die sich nur auf die Bewegung auswirkt. Diese Wartezeit wird direkt in die Bahnplanung mit aufgenommen und die Achsen mit der programmierten Dynamik abgebremst. Nach dem Ablauf der Zeit wird die folgende Bahn fortgesetzt. Der Satzvorlauf des FTL Programms wird dabei nicht angehalten.



Hinweis

Das Makro WaitOnPath eignet sich **nicht** um Anweisungen wie z. B. das Schalten eines Ausganges oder beschreiben einer Variable mit dem Programmhauptlauf zu synchronisieren.



Syntax

WaitOnPath (‹timeMS› : DINT **)**

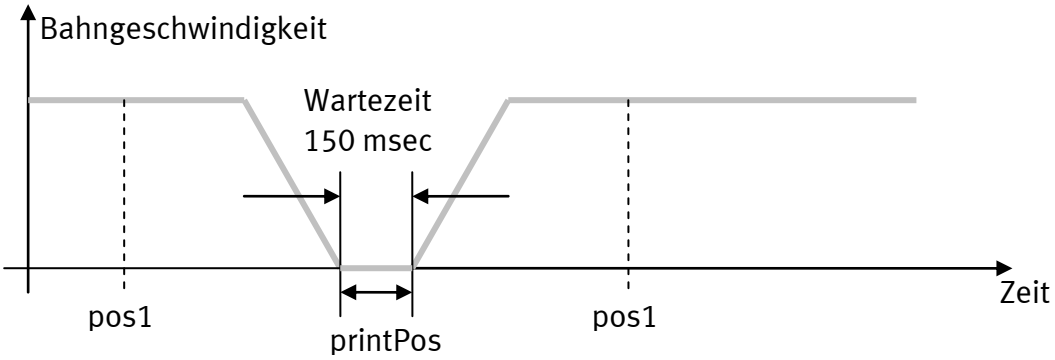
Parameter	Bedeutung	Einheit
time	Wartezeit	msec

Tabelle 8.1 Parameter Befehl WaitOnPath

Beispiel:

Mit einem Stempel wird eine Markierung auf ein Werkstück gedrückt. Nach dem Erreichen der Druckposition, muss zur Farbübertragung eine Wartezeit von 150msec ausgeführt werden. Der Satzvorlauf darf jedoch nicht angehalten werden.

```
:
Lin(pos1)                  // fahre auf Sicherheit
Lin(printPos)              // fahre auf Druckposition
WaitOnPath(150)            // warte 150 msec
Lin(pos1)                  // fahre auf Sicherheit
:
```


Bahngeschwindigkeit
Wartezeit
150 msec
Zeit
pos1
printPos
pos1

8.9.4 WaitOnPos – Anweisung mit Zeit ‹WaitOnPos›

Das Makro WaitOnPos hält den Satzvorlauf an, bis das aktuelle Segment zum angegebenen prozentualen Anteil abgearbeitet wurde. Danach wird das Programm weiter bearbeitet.

Syntax

WaitOnPos (OPT ‹pos› : REAL **)**

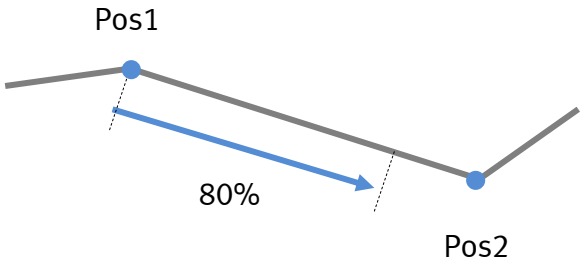
Parameter	Bedeutung	Einheit
Pos	Prozentualer Wert für Bahnsegmentlänge	Prozent

Tabelle 8.2 Parameter Befehl WaitOnPos

Ist ein Überschleifen aktiviert und die Distanz zum Zielpunkt zum Überschleifen ausreichend groß, so wird das Überschleifen ausgeführt. Der Parameter des prozentualen Anteils ist optional. Wird dieser nicht angegeben, so entspricht dies dem Wert 100%. Dies hat einen Stop auf der Bahn zur Folge.

Im folgenden Beispiel werden Position angefahren und der Satzvorlauf solange angehalten, bis 80% des Bahnsegmentes abgefahren wurde. Danach wird im Beispiel die Variable Index wird auf 10 gesetzt.

```
Lin(Pos1)
Lin(Pos2)
WaitOnPos(80)
Index := 10
```



8.10 Kommentare einfügen ‹//›

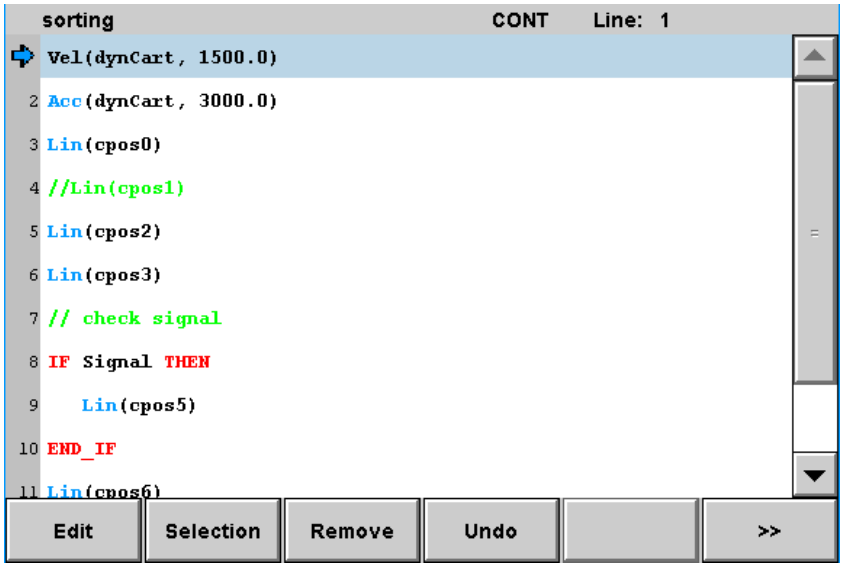
Ein Kommentar wird mit der Zeichenfolge "//" eingeleitet. Er kann ab Zeilenanfang allein oder hinter einem FTL-Befehl stehen und endet am Zeilenende

Syntax

// ‹Beliebiger Text›

Ein FTL-Befehl kann mit Kommentarzeichen "auskommentiert" werden. Dieser Befehl hat dann keine Auswirkung auf die Programmbearbeitung.

Folgende Abbildung zeigt die Programmmaske des Handbediengerätes mit einem Kommentar und einer auskommentierten Programmanweisung:



8.11 Programmzeile deaktivieren ‹##›

Mit der Zeichenfolge "##" werden Programmzeilen für die Programmbearbeitung deaktiviert, jedoch bleibt die Syntaxprüfung erhalten.

Hinweis

‹Programmanweisung›

Dies bedeutet:

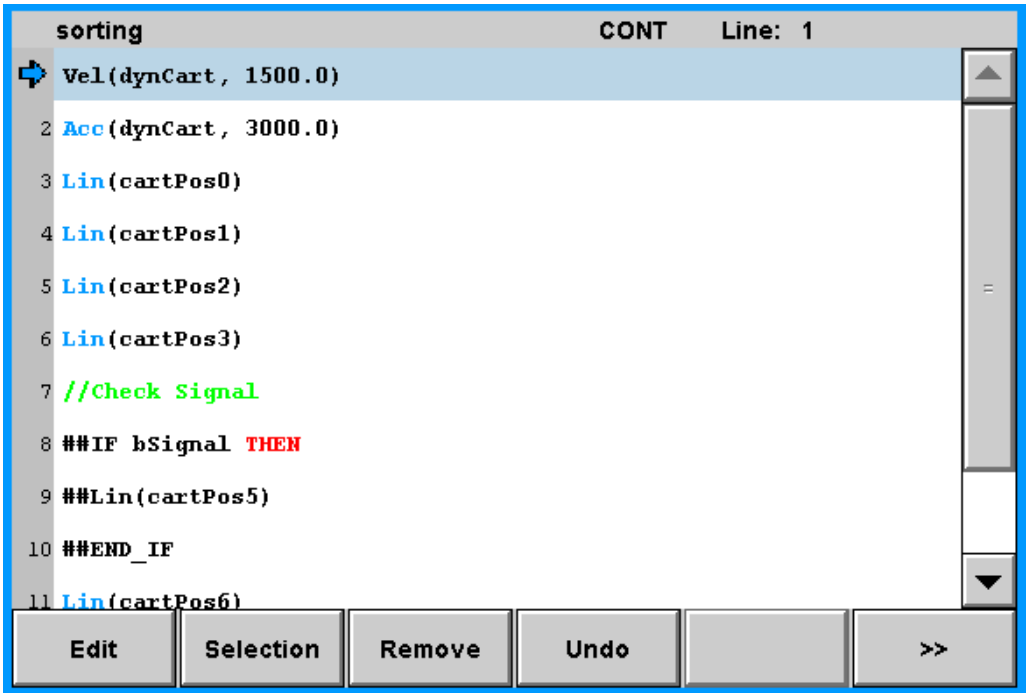
Ein deaktivierter FTL-Befehl wird im Programm nicht bearbeitet, d. h. die Inhalte haben keine Auswirkung.

Die Inhalte der Programmanweisung unterliegen der Syntaxprüfung des Compilers. Wird z. B. eine verwendete Variable gelöscht, so wird beim Programmstart ein Fehler ausgegeben.

Hinweis

Bei Programmanweisungen, die aus mehreren Zeilen bestehen, wie z. B. IF..THEN...ELSE, müssen alle zugehörigen Programmzeilen deaktiviert werden.

Folgende Abbildung zeigt die Programmmaske des Handbediengerätes mit deaktivierten Programmanweisungen in den Zeilen 8 ... 10:



9. Bewegungsbefehle

Bewegungen einer Kinematik werden mit Bewegungsbefehlen ausgelöst. Diese beschreiben die Bewegung vom aktuellen Ort zu dem angegebenen Zielpunkt. Bei der Bewegung werden zuvor gesetzte Werte wie z. B. Geschwindigkeit, Beschleunigung oder Orientierung berücksichtigt.

Bewegungen gibt es unterschiedlicher Art. Es gibt Bewegung mit einer Achsinterpolation (Point-To-Point-Bewegung) und Bewegungen in einem kartesischen Raum. Die kartesischen Bewegungen greifen dabei auf eine interne Transformation der Kinematik zurück.

9.1 Positionsbeschreibung

Alle Positionen werden in sogenannten Positionsvariablen abgespeichert. Diese Positionsvariablen werden zur Positionsangabe bei den Bewegungsbefehlen benötigt. Eine direkte Angabe mit Konstanten ist nicht möglich. Alle Positionsvariablen werden in der zugehörigen Datendatei (siehe Kapitel 3.8 FTL-Datendatei auf Seite 20) abgespeichert.

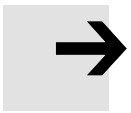
Es gibt die Möglichkeit, eine Position im Achskoordinatensystem oder in einem kartesischen Koordinatensystem anzugeben. Da diese Positionsangaben unterschiedlichen Ursprungs sind, gibt es dafür 2 Datentypen:

1. AXISPOS für die Angabe im Achskoordinatensystem,
2. CARTPOS für die Angabe im kartesischen Koordinatensystem

Die Anzahl der Achsen ist beim CMXR auf 6 begrenzt. Diese Achsen können auf Kinematische und Hilfsachsen verteilt werden. Die Positionen aller Achsen werden in einer Positionsvariablen abgespeichert.

Somit umfasst der Datentyp AXISPOS 9 Koordinatenwerte. Bedingt durch maximal 6 Freiheitsgrade umfasst der Datentyp CARTPOS ebenfalls 9 Koordinatenwerte, die sich jedoch in 3 Positionsangaben, 3 Orientierungsangaben und 3 Hilfsachsen unterteilen.

In der Programmierumgebung des FCT-PlugIn ist die Anzahl der Koordinatenwerte auf die Anzahl der projektierten Achsen begrenzt, der Rest ist gesperrt.



Hinweis

Beide Datentypen AXISPOS und CARTPOS können für Bewegungsbefehle verwendet werden. Die CMXR Steuerung führt je nach Bedarf automatisch Wandlungen (Koordinatentransformationen) aus. Entsprechendes ist der Befehlsbeschreibung zu entnehmen.

Bei den Datentypen AXISPOS und CARTPOS handelt es sich um strukturierte Datentypen (siehe Kapitel 5.5 Strukturierte Datentypen auf Seite 31).

9.1.1 Achsposition

Eine Achsposition wird mit dem strukturierten Datentyp AXISPOS beschrieben (siehe Kapitel 5.5 Strukturierte Datentypen auf Seite 31). Dieser umfasst 9 Positionen von 9 einzelnen

Achsen, deren Positionswert mit dem Datentyp REAL angegeben wird. Bedingt durch das ausgewählte Kinematikmodell kann dies eine Auswahl oder Mischung von linearen oder rotativen Achsen sein. Die Einheit einer Position wird somit in z. B. mm oder Grad angegeben.

Aufbau:

Bedingt durch die maximale Achsanzahl einer Kinematik von 9 Achsen, umfasst der Datentyp 9 Achspositionen. Diese einzelnen Positionswerte sind in 9 einzelnen REAL Variablen hinterlegt.

Datentyp AXISPOS:

a1	: REAL	Achsposition Kinematikachse 1
a2	: REAL	Achsposition Kinematikachse 2
a3	: REAL	Achsposition Kinematikachse 3
a4	: REAL	Achsposition Kinematikachse 4
a5	: REAL	Achsposition Kinematikachse 5
a6	: REAL	Achsposition Kinematikachse 6
a7	: REAL	Achsposition Hilfsachse 1
a8	: REAL	Achsposition Hilfsachse 2
a9	: REAL	Achsposition Hilfsachse 3

Der Datentyp AXISPOS beschreibt den maximal möglichen Aufbau einer Achsposition. Sind Achsen nicht vorhanden, so ist eine Positionsangabe für diese Achsen irrelevant. In der Regel erfolgt für diese Achsen die Angabe des Positionswertes 0. Im FCT-Editor sind die Felder dieser Achsen gesperrt.

Auf jeden einzelnen Wert eines Positionswertes vom Typ AXISPOS kann über den Namen der einzelnen Position zugegriffen werden.

Beispiel:

Variable:

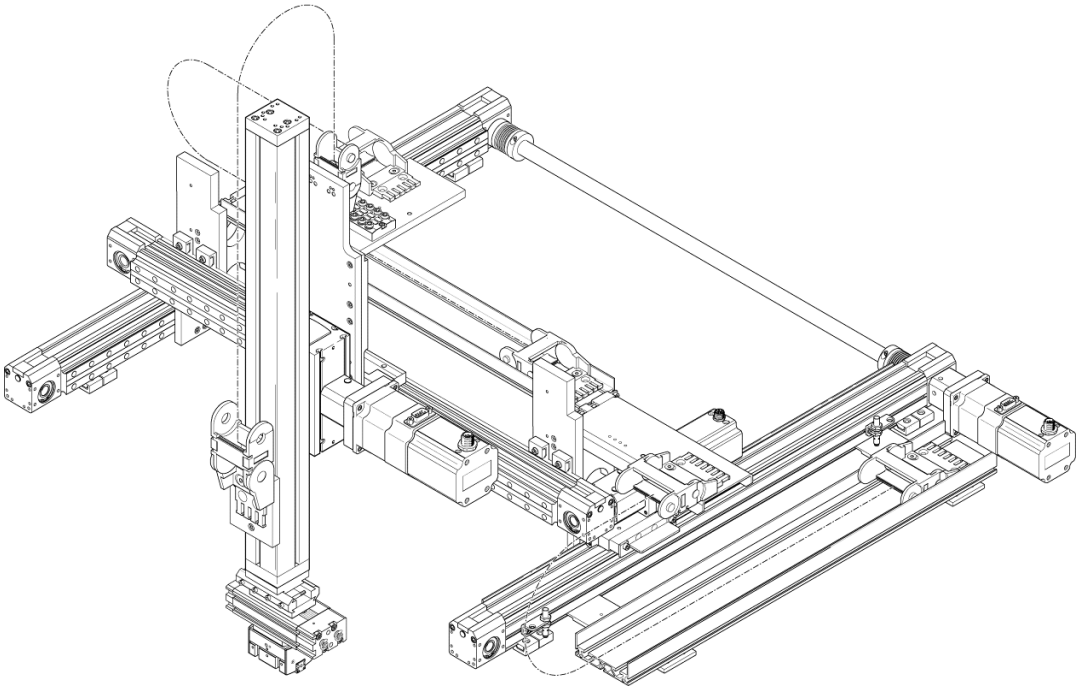
```
startPos : AXISPOS := (100, 50, 30, 0, 0, 0, 0, 0, 0)
posA1    : REAL
posA2    : REAL
```

Programm:

```
:
Lin(startPos)          // fahren auf startPos
posA1 := startPos.a1   // umspeichern von Achswert 1
posA2 := startPos.a2   // umspeichern von Achswert 2
:
```

Die Zuordnung einzelner Achsen auf die Positionsdatenstruktur AXISPOS erfolgt durch eine Nummerierung. Diese Nummerierung hat bereits in der Konfiguration der Kinematikachsen stattgefunden. Analog zu dieser Nummerierung erfolgt bei der Positionsangabe des Datentyps AXISPOS die Zuordnung zu den einzelnen Achsen.

Beispiel:



Kartesisches Raumportal mit 3 Linearachsen und einer Rotationsachse (Drehen des Greifers). In der Konfiguration erfolgte folgende Festlegung:

Achse 1 = X-Achse

Achse 2 = Y-Achse

Achse 3 = Z-Achse

Achse 4 = Drehachse Greifer

Achse 5 bis 6 sind nicht vorhanden, keine Hilfsachsen.

Analog zu dieser Nummerierung erfolgt die Zuordnung im Datentyp Axispos:

a1 : REAL Position Achse 1 = X - Achse

a2 : REAL Position Achse 2 = Y - Achse

a3 : REAL Position Achse 3 = Z - Achse

a4 : REAL Position Achse 4 = Drehachse Greifer

a5 : REAL Position Achse 5, nicht vorhanden

a6 : REAL Position Achse 6, nicht vorhanden

a7 : REAL Position Hilfsachse 1, nicht vorhanden

a8 : REAL Position Hilfsachse 2, nicht vorhanden

a9 : REAL Position Hilfsachse 3, nicht vorhanden

9.1.2 Kartesische Position

Im Gegensatz zur Achsposition des Datentyps AXISPOS beschreibt der kartesische Positionstyp CARTPOS eine Position in einem kartesischen Koordinatensystem.

Ein Körper besitzt maximal 6 Freiheitsgrade. Mit Hilfe dieser Werte kann man die Position und Orientierung des Körpers im Raum bestimmen. Diese 6 Freiheitsgrade werden mit 6 Angaben im Datentyp CARTPOS beschrieben. Zusätzlich erfolgt die Angabe der Position der 3 Hilfsachsen. Diese ist jedoch eine Angabe in einer Achsposition, da mit den Hilfsachsen nicht kartesisch verfahren werden kann. Sie werden gemeinsam mit den Kinematikachsen auf die Zielposition interpoliert, jedoch führen die Hilfsachsen eine Point-To-Point (Ptp) Interpolation aus.

Aufbau:

Datentyp CARTPOS

x	: REAL	Verschiebung entlang der X-Achse
y	: REAL	Verschiebung entlang der Y-Achse
z	: REAL	Verschiebung entlang der Z-Achse
a	: REAL	Orientierungsangabe, Drehung um die Z-Achse
b	: REAL	Orientierungsangabe, Drehung um die verdrehte Y-Achse
c	: REAL	Orientierungsangabe, Drehung um die verdrehte Z-Achse
aux1	: REAL	Achsposition Hilfsachse 1
aux2	: REAL	Achsposition Hilfsachse 2
aux3	: REAL	Achsposition Hilfsachse 3

Bedingt durch die ausgewählte Kinematik bzw. deren Freiheitsgrade, ist manche Angabe in einer kartesischen Position nicht möglich. Die einzelnen Positionsangaben im Datentyp CARTPOS sind nicht an die physikalischen Achsen der Kinematik gebunden, sondern an deren Freiheitsgrade.

Die kartesische Position umfasst die Angaben für maximal 6 Freiheitsgrade. X, Y, Z sind die translatorischen Positionen, A, B und C beschreiben die Orientierungen der Position. Die Orientierung wird, wie im gesamten System, nach der Euler ZYZ Konvention angegeben.

Der Datentyp CARTPOS ist wie der Datentyp AXISPOS strukturiert (siehe Kapitel 5.5 Strukturierte Datentypen auf Seite 31). Der Zugriff auf einzelne Positionswerte erfolgt über den Namen der einzelnen Werte des Datentyps.

Beispiel:

Variable:

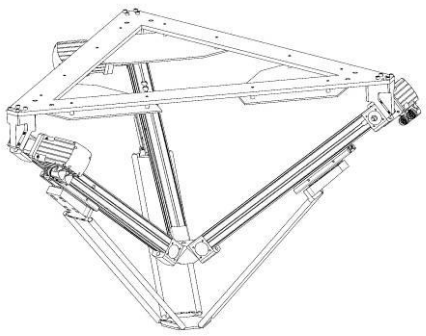
```
startPos : CARTPOS := (1050, 130, 30, 0, 0, 0, 0, 0, 0)
newPos   : CARTPOS := (0, 0, 0, 0, 0, 0, 0, 0, 0)
posX     : REAL
posY     : REAL
```

Programm:

```
:
Lin(startPos)                  // fahren auf StartPos
newPos    := startPos          // kopieren von StartPos
newPos.x := newPos.x + 10      // berechnen von X
newPos.y := newPos.y + 35.7    // berechnen von Y
Lin(newPos)                    // fahre auf berechnete Position
:
```

Die Wirkung der einzelnen kartesischen Positions- und Orientierungsangaben hängt von den Freiheitsgraden der Kinematik ab.

Beispiel:



Gegeben ist eine Parallelstab-Kinematik (Tripod) mit 3 Hauptachsen, ohne Handachsen. Mit den vorhandenen 3 Hauptachsen werden die 3 translatorischen Freiheitsgrade X, Y und Z abgedeckt. Da keine Handachsen vorhanden sind, ist keine Orientierung des Werkzeuges möglich. Eine Programmierung der Variablen a, b, c, aux1, aux2 oder aux3 in der Positionsangabe des Datentyps CARTPOS hat keine Auswirkungen.

9.2 Point-To-Point-Bewegung ‹Ptp›

Die Point-To-Point-Bewegung (PTP) ist die schnellste Möglichkeit, die Werkzeugspitze (TCP) an die gewünschte Position zu bewegen. Die PTP-Bewegung ist ein synchroner Punkt-zu-Punkt-Fahrbefehl mit Zielposition. Bei diesem Befehl werden alle Achsen gleichzeitig gestartet und gelangen gleichzeitig an die programmierte Zielposition. Zur Bewegung werden die gerade aktiven Dynamikwerte verwendet, z. B. Geschwindigkeit und Beschleunigung. Die effektive Dynamik ergibt sich aus Kombination der Dynamiken aller beteiligten Achsen. Die langsamste Achse bestimmt die Dynamik. Die Bewegung des TCP ergibt sich in diesem Fall aus der Kombination der Einzelachsbewegung. Die Bewegung am TCP ist undefiniert.

Vorsicht

- Da bei einer PTP-Bewegung keine Bahn eingehalten wird, sondern nur die Achsen auf das Ziel interpoliert werden, kann es zu nicht einschätzbaren Beschleunigungen und Geschwindigkeiten am Werkzeug (TCP) kommen. Die Bewegungen sind daher auf Verlust oder Beschädigung von Werkstück und/oder Werkzeug zu prüfen.

- Aufgrund der Achsinterpolation können bei einer PTP-Bewegung keine Werkzeugdaten berücksichtigt werden. Deshalb ist stets auf die eingesetzten Werkzeuge zu achten und diese vor Beschädigungen zu schützen.

- Um die Kollisionsgefahr zu minimieren, sollten alle PTP-Bewegungen mit einer reduzierten Geschwindigkeit getestet werden. Z. B. kann dazu der Override verwendet werden.

Syntax

Ptp (‹Pos› : AXISPOS oder CARTPOS)

Parameter	Bedeutung	Einheit
Pos	Zielposition	AXISPOS oder CARTPOS

Tabelle 9.1 Parameter Befehl PTP

Die Positionsangabe kann kartesisch oder im Achskoordinatensystem (bezogen auf jede einzelne Achse) sein. Die CMXR Mehrachssteuerung transformiert die Positionen entsprechend.

Beispiel:

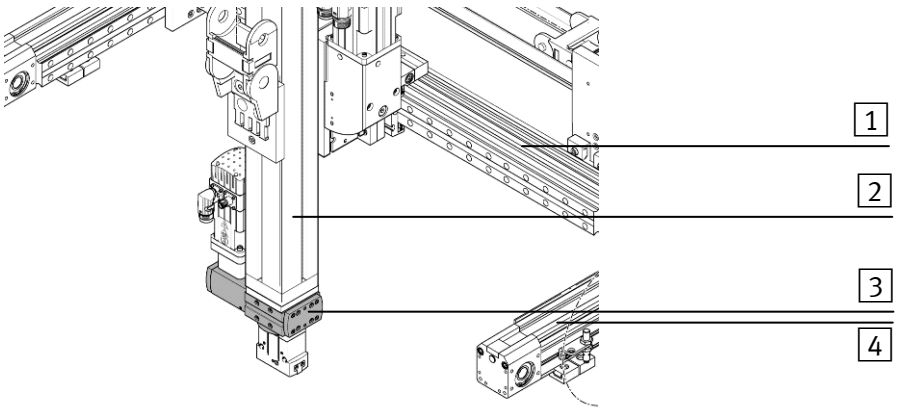
Ein kartesisches Portal mit 3 Achsen X, Y, Z und einer Drehachse mit Greifer muss positioniert werden.

1 Y-Achse

2 Z-Achse

3 Drehachse mit Greifer

4 X-Achse

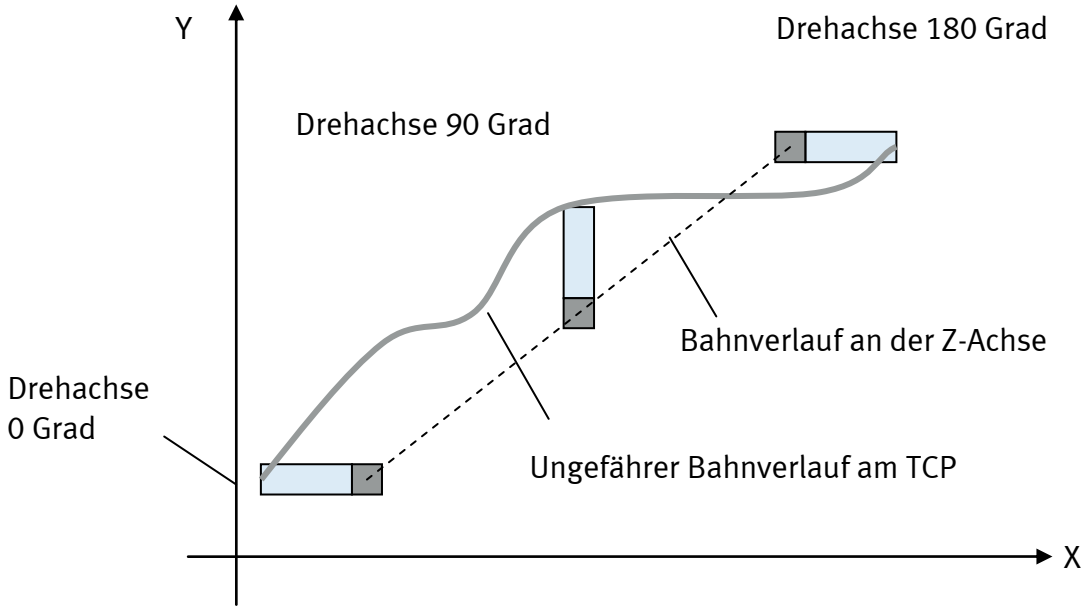


Variable:

```
pos1 : CARTPOS := (100, 50, 100, 0, 0, 0, 0, 0, 0)
pos2 : CARTPOS := (600, 550, 100, 180, 0, 0, 0, 0, 0)
```

Programm:

```
:
Ptp(pos1)
Ptp(pos2)
:
```



Im Beispiel ist der Bahnverlauf der Z-Achse und der Werkzeugspitze (TCP) ersichtlich. Durch die senkrechte Anordnung der kartesischen Kinematik ist der Bahnverlauf an der Z-Achse eine Gerade. Dadurch, dass das Werkzeug einen Versatz zur Z-Achse besitzt, kommt es bei dessen Bahnverlauf zu einem nicht vorhersehbaren Verhalten, da mit der synchronen Interpolation alle Achsen gemeinsam auf ihren Zielpunkt fahren, ohne Rücksicht auf den Verlauf der gefahrenen Bahn.

9.3 Relative Point-To-Point-Bewegung ‹PtpRel›

Analog zum PTP-Befehl funktioniert der relative PTP-Befehl. Mit dem Unterschied, dass die angegebene Position relativ zur Startposition erfolgt. Die Positionsangabe wird zur Startposition addiert.

Eine Anwendung wäre z. B. eine relative Positionierung innerhalb eines Rasters wie bei einer Palette.

Syntax

PtpRel (‹Dist› : AXISDIST oder CARTDIST)

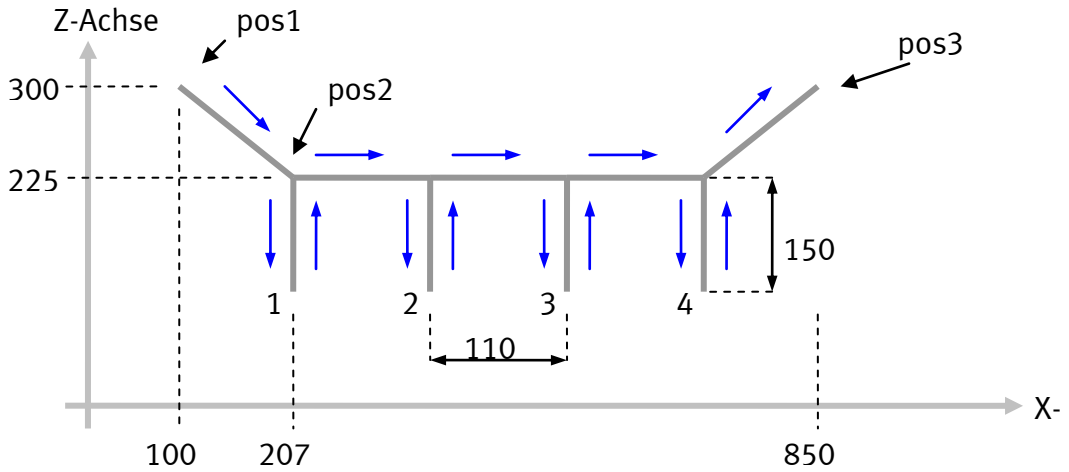
Parameter	Bedeutung	Einheit
Dist	Relative Entfernung, die gefahren werden soll	AXISDIST oder CARTDIST

Tabelle 9.2 Parameter Befehl PtpRel

Die Distanzangabe kann kartesisch oder bezogen auf jede einzelne Achse sein. Die CMXR-Steuerung transformiert die Positionen entsprechend.

Beispiel:

In einer Handhabungsapplikation wird ein Teil an 4 Positionen jeweils an eine Messeinheit angefahren.



Variable:

```
:
pos1      : CARTPOS := (100, 0, 300, 0, 0, 0, 0, 0, 0)
pos2      : CARTPOS := (207, 0, 225, 0, 0, 0, 0, 0, 0)
pos3      : CARTPOS := (850, 0, 300, 0, 0, 0, 0, 0, 0)
distX     : CARTDIST := (110, 0, 0, 0, 0, 0, 0, 0, 0)
distZpos  : CARTDIST := (0, 0, 150, 0, 0, 0, 0, 0, 0)
distZneg  : CARTDIST := (0, 0, -150, 0, 0, 0, 0, 0, 0)
:
```

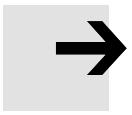
Programm:

```
:
Ptp(pos1)
Ptp(pos2)
LOOP 3 DO
  PtpRel(distZneg)
  CALL Check()       // Aufruf Pruefzyklus
  PtpRel(distZpos)
  PtpRel(distX)
END_LOOP
PtpRel(distZneg)
CALL Check()         // Aufruf Pruefzyklus
PtpRel(distZpos)
PtpRel(distX)
Ptp(pos3)
:
```

Der Messzyklus für das Werkstück, sowie die Auswertung werden im Unterprogramm „Pruefe“ abgehandelt. Der Inhalt des Unterprogramms ist zur Vereinfachung nicht dargestellt.

9.4 Bewegung einer Achse, ‹MoveAxisPtp›, ‹MoveAxisCart›

Mit dem Befehl **MoveAxisPtp** und **MoveAxisCart** kann eine Achse der Kinematik mit einer PTP-Bewegung oder kartesischen Bewegung positioniert werden. Die Zielposition der Achse wird absolut angegeben.



Syntax

MoveAxisPtp (‹Axis› : ENUM, ‹Pos› : REAL)

MoveAxisCart (‹CartComp› : ENUM, ‹Pos› : REAL)

Parameter	Bedeutung	Einheit
Axis	Ausgewählte physikalische Achse, die verfahren werden soll.	Enumeration mit der Wertigkeit A1 bis A9 für Achse 1 bis 9
Pos	Absolute Zielposition	Einheit der definierten Achse

Tabelle 9.3 Parameter Befehl MoveAxisPtp

Parameter	Bedeutung	Einheit
CartComp	Ausgewählte kartesische Achse, die verfahren werden soll.	Enumeration mit der Wertigkeit X Y Z ; A B C
Pos	Absolute kartesische Zielposition	Einheit der definierten Achse

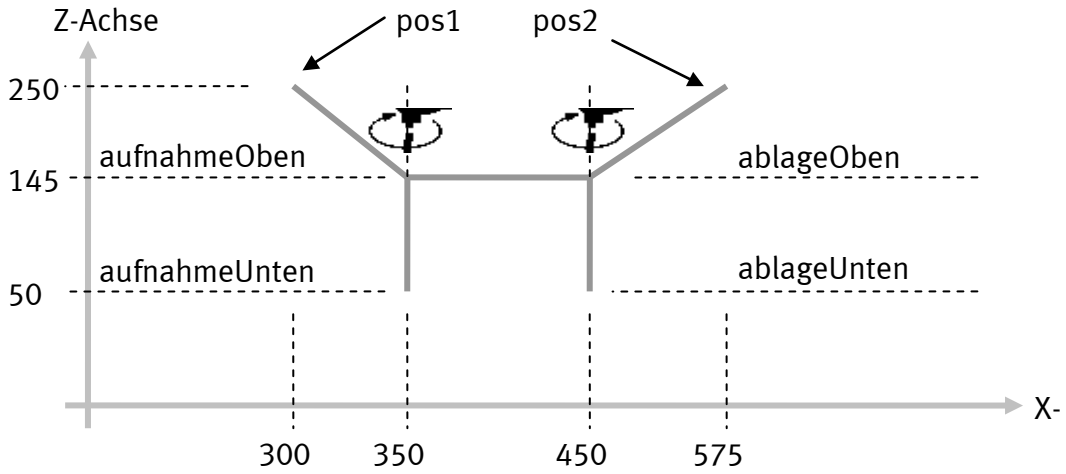
Tabelle 9.4 Parameter Befehl MoveAxisCart

Da es sich hier um eine Einzelachse handelt, erfolgt die Bewegung unter Berücksichtigung etwaiger Begrenzungen, z. B. Override, mit voller Achsdynamik.

Beispiel:

Eine kartesische Kinematik besteht aus 4 Achsen:
- Achse 1 = X-Achse,
- Achse 2 = Y-Achse,
- Achse 3 = Z-Achse und
- Achse 4 = Drehachse des Werkzeuges.

Im Beispiel muss ein Werkstück an einer Position aufgenommen und an einer anderen Position abgelegt werden. Dazu muss die Drehachse entsprechend positioniert werden.



Variable:

```
:
pos1       : CARTPOS := (300, 0, 250, 0, 0, 0, 0, 0, 0)
takeAbove  : CARTPOS := (350, 0, 145, 0, 0, 0, 0, 0, 0)
pos2       : CARTPOS := (575, 0, 250, 0, 0, 0, 0, 0, 0)
takePos    : REAL       := 96.5
:
```

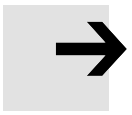
Programm mit PTP-Bewegungen:

```
:
Ptp(pos1)                  // anfahren
Ptp(takeAbove)             // Aufnahme oben
MoveAxisPtp(A4, takePos)   // Greifer drehen
MoveAxisPtp(A3, 50)        // nach unten
Gripper.Set()              // Greifer schließen
MoveAxisPtp(A3, 145)       // nach oben
MoveAxisPtp(A1, 450)       // Ablage oben
MoveAxisPtp(A4, 180)       // drehen auf Ablage
MoveAxisPtp(A3, 50)        // Ablage unten
Gripper.Reset()            // Greifer öffnen
MoveAxisPtp(A3, 145)       // Ablage oben
Ptp(pos2)                  // abfahren
:
```

9.5 Linearbewegung ‹Lin›

Bei einer Linearbewegung berechnet die CMXR Mehrachssteuerung eine Gerade, die von der momentanen Position (Startposition) zur programmierten Position (Zielposition) führt. Diese Bewegung wird unter Berücksichtigung der gesetzten Bahnwerte wie z. B. Bahnbeschleunigung, Bahngeschwindigkeit, Orientierungen und Werkzeugdaten berechnet und ausgeführt. Wird in der Zielpositionsangabe eine Orientierungsänderung angegeben, so wird auf dieser Bahn kontinuierlich von der Anfangsorientierung am Startpunkt auf die Endorientierung verfahren.

Die Linearbewegung ist eine kartesische Bewegung, d. h. die Bewegung wird mit Hilfe der intern implementierten Koordinatentransformation für die vorliegende Kinematik berechnet. Dabei wird stets die Position an der Werkzeugspitze (TCP) programmiert. Auch alle Dynamikwerte wie Bahnbeschleunigung und Bahngeschwindigkeit werden direkt am TCP erreicht. Dies hat den Vorteil, dass die Dynamikwerte am Werkzeug begrenzt und bekannt sind. Auf die Greifereinheit wirken somit reproduzierbare Kräfte.



Syntax

Lin (‹Pos› : AXISPOS oder CARTPOS)

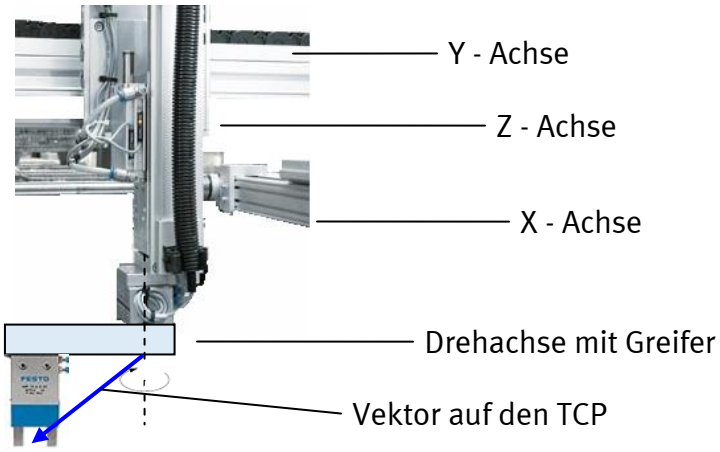
Parameter	Bedeutung	Einheit
Pos	Absolute Zielposition	AXISPOS oder CARTPOS

Tabelle 9.5 Parameter Befehl Lin

Die Positionsangabe kann kartesisch oder bezogen auf jede einzelne Achse sein. Die CMXR Mehrachssteuerung transformiert die Positionen entsprechend.

Beispiel:

Ein kartesisches Portal mit 3 Achsen X, Y, Z und einer Drehachse am Greifer muss positioniert werden. Die Werkzeugspitze (TCP) ist mit einem Vektor auf den Mittelpunkt des Greifers definiert (siehe Kapitel 14 Werkzeuge auf Seite 122).

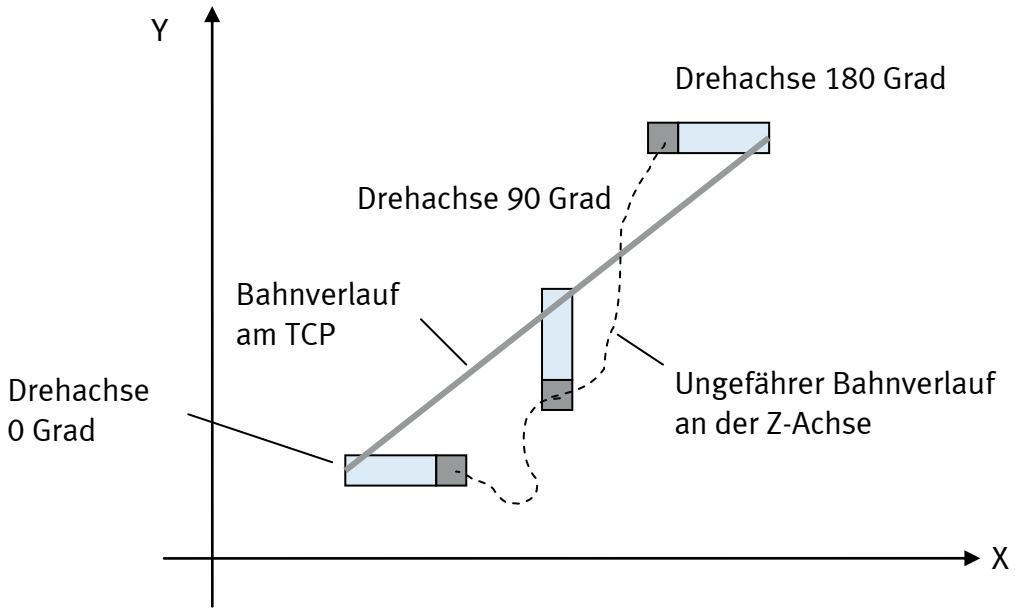


Variable:

```
:
pos1    : CARTPOS := (100, 50, 100, 0, 0, 0, 0, 0, 0)
pos2    : CARTPOS := (600, 550, 100, 180, 0, 0, 0, 0, 0)
gripper : TCPTOOL := (-100, 0, 97, 0, 0, 0,)
:
```

Programm:

```
:
Tool(gripper)
Lin(pos1)
Lin(pos2)
:
```



Wie dargestellt, wird bei einer Linearbewegung der TCP (Werkzeugspitze) auf der Bahn geführt. Alle Geschwindigkeitsangaben beziehen sich dabei immer auf den TCP, somit ist der Bahnverlauf bestimmt. Nicht vorhersehbar ist jedoch der Verlauf der Bahn des Werkzeugflansches in der X-Y- Ebene. Dieser Bahnverlauf ergibt sich aus der Konstellation der Kinematik und des Vektors zum TCP und wird durch die interne Koordinatentransformation berechnet.

9.6 Relative Linearbewegung ‹LinRel›

Analog zum Lin-Befehl funktioniert der relative Lin-Befehl. Hier wird die Positionsangabe zur Startposition addiert.

Eine Anwendung wäre z. B. eine relative Positionierung innerhalb eines Rasters wie bei einer Palette.

Syntax

LinRel (‹Dist› : AXISDIST oder CARTDIST)

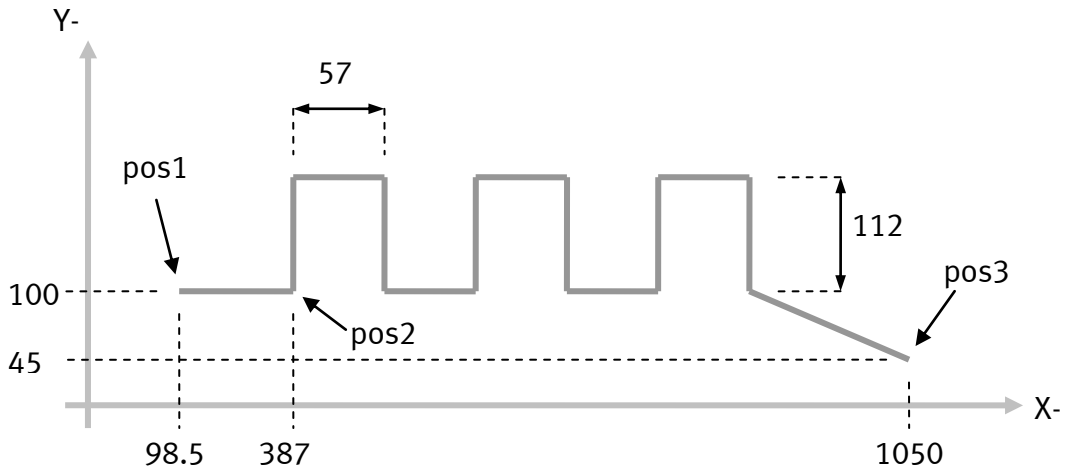
Parameter	Bedeutung	Einheit
Dist	Relative Zielposition	AXISDIST oder CARTDIST

Tabelle 9.6 Parameter Befehl LinRel

Die Distanzangabe kann kartesisch oder bezogen auf jede einzelne Achse angegeben werden. Die CMXR Mehrachssteuerung transformiert die Positionen entsprechend.

Beispiel:

Eine Kontur enthält sich wiederholende Bahnstücke. Diese Bahnstücke lassen sich relativ beschreiben. Mit dem Einsatz einer Schleifenprogrammierung lässt sich eine solche Applikation komfortabel lösen.



Variable:

```
:
pos1      : CARTPOS := (98.5, 100, 0, 0, 0, 0, 0, 0, 0)
pos2      : CARTPOS := (387, 100, 0, 0, 0, 0, 0, 0, 0)
distX     : CARTDIST := (57, 0, 0, 0, 0, 0, 0, 0, 0)
distYpos  : CARTDIST := (0, 112, 0, 0, 0, 0, 0, 0, 0)
distYneg  : CARTDIST := (0, -112, 0, 0, 0, 0, 0, 0, 0)
pos3      : CARTPOS := (1050, 45, 0, 0, 0, 0, 0, 0, 0)
direction : BOOL
:
```

Programm:

```
:
Lin(pos1)
Lin(pos2)
direction := TRUE
LOOP 5 DO
  IF direction = TRUE THEN
    LinRel(distYpos)
  ELSE
    LinRel(distYneg)
  END_IF
  LinRel(distX)
  direction := NOT direction
END_LOOP
LinRel(distYneg)
Lin(pos3)
:
```

Mit der Variablen „direction“ wird die Richtung der relativen Bewegung der Y-Achse festgelegt. Auf diese Weise lässt sich die Bewegung mit einer Schleife programmieren.

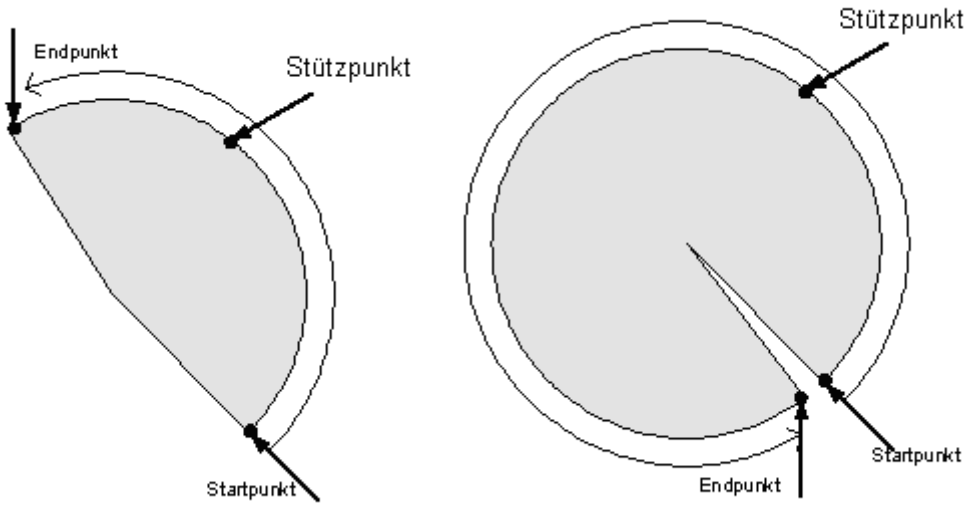
9.7 Zirkularbewegung mit Stützpunkt

Die Kreisinterpolation unterscheidet sich von der Linearinterpolation nicht nur durch die geometrische Form, sondern auch dadurch, dass neben Start- und Endpunkt zusätzlich ein Stützpunkt angegeben werden muss, damit der Kreis eindeutig bestimmt ist.

9.7.1 Funktionsweise

Die Kreisdefinition erfolgt mit einem Stützpunkt, der auf der Kreisbahn liegen muss, und dem Endpunkt der Kreisbahn. Bei der Kreisbahn wird zuerst der Stützpunkt, dann der Endpunkt angefahren. Der Radius der Kreisbahn ergibt sich aus einer internen Berechnung mit Startpunkt, Stützpunkt und Endpunkt der Kreisbahn.

Folgende Abbildung zeigt eine Zirkularbewegung mit Verwendung eines Stützpunktes:



Der Kreis wird immer so gefahren, dass sich der TCP vom Startpunkt über den Stützpunkt zum Endpunkt bewegt. Der Stützpunkt liegt per Definition immer zwischen dem Start- und Endpunkt.

Einschränkungen:

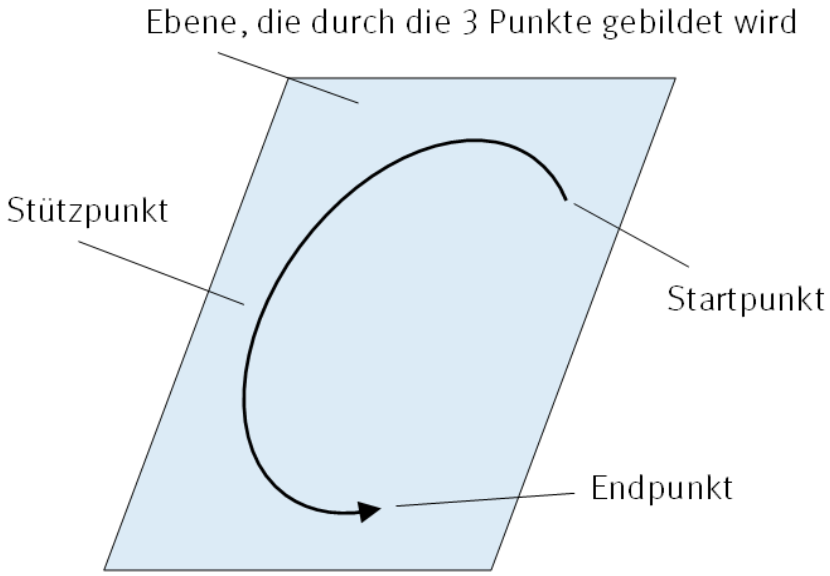
Eine Fehlermeldung erhält man, wenn zumindest zwei der den Kreis definierenden Positionen die gleiche Position haben, oder wenn alle Punkte auf einer Geraden liegen. In diesen Fällen lässt sich die Kreisbahn mathematisch nicht berechnen.

Mit dieser Methode kann kein Vollkreis (360°) beschrieben werden. Deshalb muss ein Vollkreis aus 2 Halbkreisen zusammengesetzt werden.

Die Orientierung des Stützpunktes wird zur Interpolation des Kreisbogens nicht berücksichtigt. Die Interpolation erfolgt ausschließlich zwischen Start- und Endpunkt. Ist es erforderlich, Orientierungen innerhalb eines Kreisbogens zu wechseln, so besteht die Möglichkeit, einen Kreisbogen in mehrere Teile zu zerlegen, um dann an den Start- und Endpunkten die Orientierungen vorzugeben.

9.7.2 Ebenendefinition

Die Kreisbahn wird auf einer Ebene gefahren, die anhand der 3 Punkte Startpunkt, Stützpunkt und Endpunkt definiert wird. Durch diese Definition spannt sich die Ebene im Raum auf, auf der die Kreisbahn gefahren wird.



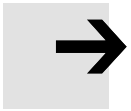
Die Abbildung zeigt eine Kreisbahn mit ihren 3 Stützpunkten, die eine Ebene im Raum definieren, auf der die Kreisbahn liegt.

Hinweis

Die Kreisbahn liegt immer auf einer Ebene, eine Schraubeninterpolation, mit einer zusätzlichen Interpolation senkrecht zur Ebene ist nicht möglich.

9.7.3 Zirkularbefehl mit Stützpunkt ‹CircIp›

Der Zirkularbefehl mit Stützpunkt hat folgende Programmsyntax:



Syntax

CircIp (‹IpPos› : AXISPOS oder CARTPOS,
‹Pos› : AXISPOS oder CARTPOS)

Parameter	Bedeutung	Einheit
IpPos	Stützpunkt auf dem Kreis	AXISPOS oder CARTPOS
Pos	Endposition des Kreisbogens	AXISPOS oder CARTPOS

Tabelle 9.7 Parameter Befehl CircIp

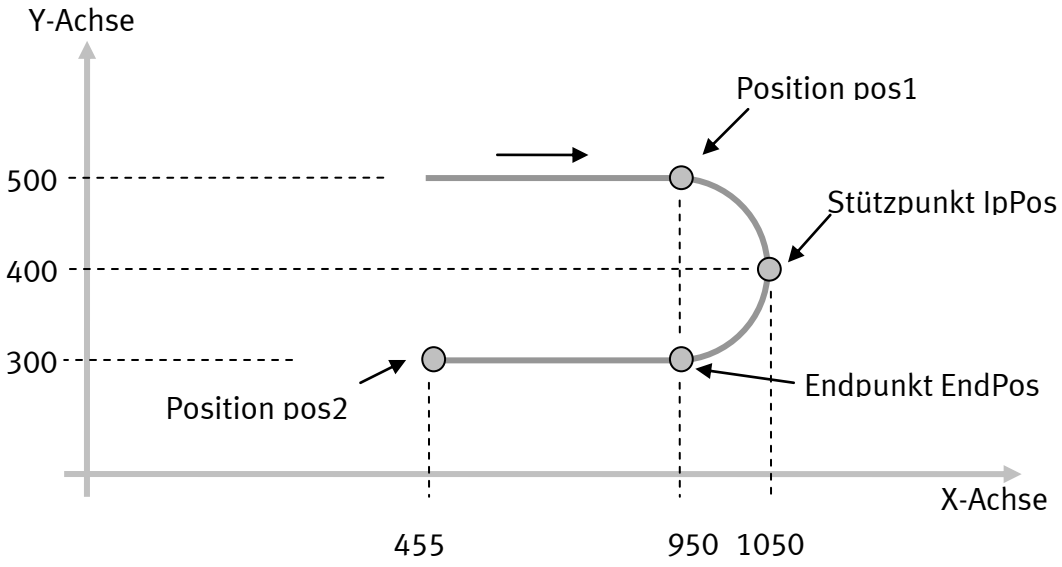
Die Positionsangaben können kartesisch oder bezogen auf jede einzelne Achse sein. Die CMXR Mehrachssteuerung transformiert die Positionen entsprechend.

Warnung

Der Kreisbefehl benötigt die Angabe des Stütz- und Endpunktes. Der Startpunkt wird durch den Endpunkt der vorherigen Bewegung gebildet. Wird dieser Punkt verschoben, kommt es zu einer Veränderung der Kreisbahn. Diese könnte ungewollt sein und könnte zu einer Kollision führen. Die Änderung des Startpunktes muss zu keiner Fehlermeldung führen, weil das Ergebnis rechnerisch korrekt ist.

Beispiel:

Mit einer Kinematik soll folgende Kontur abgefahren werden:



Die Z-Achse bleibt auf der Koordinate 0. Der Startpunkt des Kreisbogens wird über einen separaten Fahrbefehl z. B. **Ptp** oder **Lin** angefahren.

Variablen:

```
:
pos1    : CARTPOS := (950, 500, 0, 0, 0, 0, 0, 0, 0)
pos2    : CARTPOS := (455, 300, 0, 0, 0, 0, 0, 0, 0)
IpPos   : CARTPOS := (1050, 400, 0, 0, 0, 0, 0, 0, 0)
EndPos  : CARTPOS := (950, 300, 0, 0, 0, 0, 0, 0, 0)
:
```

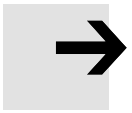
Programm :

```
:
Lin(pos1)                       // Anfahren Startpunkt
CircIp(IpPos, EndPos)           // Kreisbewegung auf Endpunkt
Lin(pos2)                       // Abfahren
:
```

9.7.4 Zirkularbefehl mit Stützpunkt, PTP-Anfahrt ‹PtpToCircIp›

Im Gegensatz zum Zirkularbefehl **CircIp** besitzt dieser Befehl den Startpunkt des Kreisbogens in der Parameterliste. Dies hat den Vorteil, dass der Kreisbogen konsistent beschrieben wird. Der Startpunkt des Kreisbogens wird mit einem PTP-Befehl angefahren. Sonst ist das Verhalten gleich wie beim CircIp-Befehl.

Da es sich beim Anfahren des Startpunktes um eine PTP-Bewegung und bei der Kreisinterpolation um eine kartesische Bewegung handelt, ist kein geometrisches Überschleifen und keine konstante Bahngeschwindigkeit möglich. Das Überschleifen wird anhand den Möglichkeiten einer PTP-Bewegung durchgeführt.



Syntax

PtpToCircIp (‹StartPos› : AXISPOS oder CARTPOS,
‹IpPos› : AXISPOS oder CARTPOS,
‹Pos› : AXISPOS oder CARTPOS)

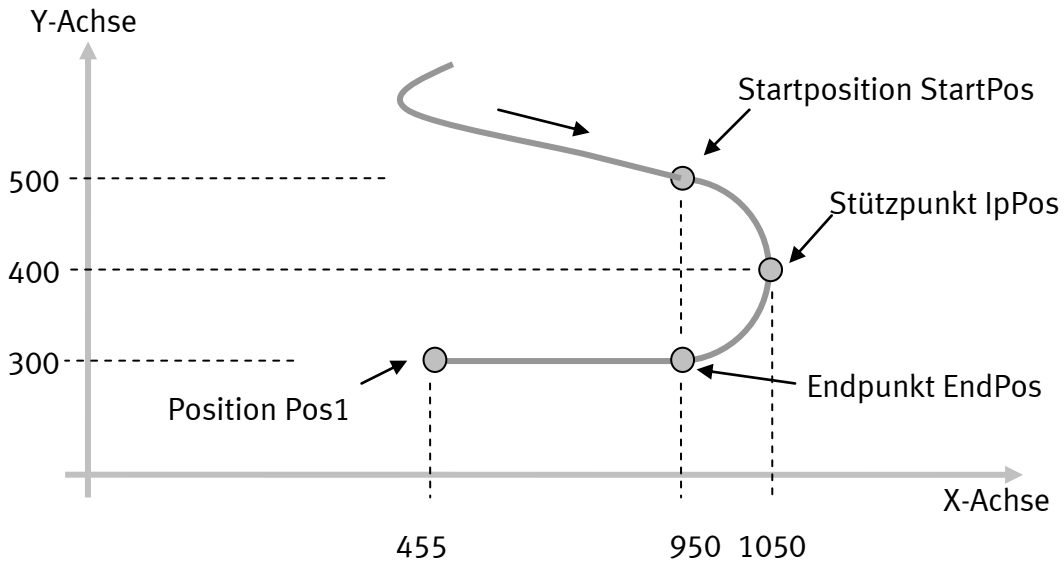
Parameter	Bedeutung	Einheit
StartPos	Startpunkt des Kreisbogens	AXISPOS oder CARTPOS
IpPos	Stützpunkt des Kreisbogens	AXISPOS oder CARTPOS
Pos	Endpunkt des Kreisbogens	AXISPOS oder CARTPOS

Tabelle 9.8 Parameter Befehl PtpToCircIp

Die Positionsangaben können kartesisch oder bezogen auf jede einzelne Achse sein. Die CMXR Mehrachssteuerung transformiert die Positionen entsprechend.

Beispiel:

Mit einer Kinematik soll folgende Kontur abgefahren werden:



Die Ebene für die Z-Achse beträgt 0. Der Startpunkt des Kreisbogens wird mit einer PTP (Point-To-Point) Bewegung angefahren. Der Startpunkt wird dem Befehl **PtpToCircIp** übergeben.

Variablen:

```
:
StartPos : CARTPOS := (950, 500, 0, 0, 0, 0, 0, 0, 0)
IpPos    : CARTPOS := (1050, 400, 0, 0, 0, 0, 0, 0, 0)
EndPos   : CARTPOS := (950, 300, 0, 0, 0, 0, 0, 0, 0)
Pos1     : CARTPOS := (455, 300, 0, 0, 0, 0, 0, 0, 0)
:
```

Programm:

```
:
// PTP Anfahren, kartesische Kreisbewegung
PtpToCircIp(StartPos, IpPos, EndPos)
Lin(Pos1)                        // Abfahren
:
```

9.7.5 Zirkularbefehl mit Stützpunkt, Lin Anfahrt ‹LinToCircIp›

Wie der Befehl **PtpToCircIp** beinhaltet der Befehl **LinToCircIp** den Startpunkt der Kreisbahn. Die Anfahrt der Kreisbahn erfolgt nun kartesisch. Dies bedeutet, dass die Bewegung auf den Startpunkt und die Kreisbahn im Falle des Überschleifens geometrisch erfolgen kann. Eine konstante Bahngeschwindigkeit ist ebenso möglich.

Syntax

LinToCircIp (‹StartPos› : AXISPOS oder CARTPOS,
‹IpPos› : AXISPOS oder CARTPOS,
‹Pos› : AXISPOS oder CARTPOS)

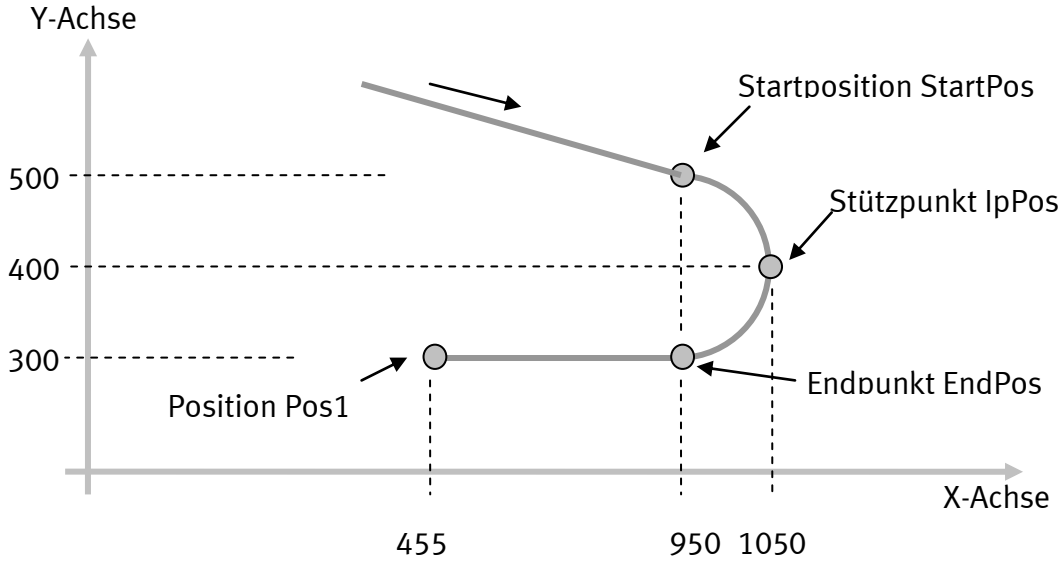
Parameter	Bedeutung	Einheit
StartPos	Startpunkt des Kreisbogens	AXISPOS oder CARTPOS
IpPos	Stützpunkt des Kreisbogens	AXISPOS oder CARTPOS
Pos	Endpunkt des Kreisbogens	AXISPOS oder CARTPOS

Tabelle 9.9 Parameter Befehl LinToCircIP

Die Positionsangaben können kartesisch oder bezogen auf jede einzelne Achse sein. Die CMXR Mehrachssteuerung transformiert die Positionen entsprechend.

Beispiel:

Mit einer Kinematik soll folgende Kontur abgefahren werden:



Die Ebene für die Z-Achse beträgt 0. Der Startpunkt des Kreisbogens wird mit einer Linearbewegung angefahren. Der Startpunkt wird dem Befehl **LinToCirc** übergeben.

Variablen:

```
:
StartPos : CARTPOS := (950, 500, 0, 0, 0, 0, 0, 0, 0)
IpPos    : CARTPOS := (1050, 400, 0, 0, 0, 0, 0, 0, 0)
EndPos   : CARTPOS := (950, 300, 0, 0, 0, 0, 0, 0, 0)
Pos1     : CARTPOS := (455, 300, 0, 0, 0, 0, 0, 0, 0)
:
```

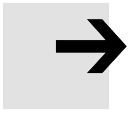
Programm:

```
:
// Lin Anfahren, kartesische Kreisbewegung
LinToCircIp(StartPos, IpPos, EndPos)
Lin(Pos1)                          // Abfahren
:
```

9.8 Stoppen der Bewegung ‹StopMove›

Mit dem Befehl **StopMove** wird die Kinematik gestoppt und alle bereits berechneten Bahndaten werden verworfen. Der Befehl wirkt sich im Satzvorlauf aus.

Das Stoppen der Kinematik erfolgt mit der maximalen Bremsrampe, die für das Stoppen der Kinematik definiert ist. Reduktionen der Dynamik durch einen Override wirken sich beim Stoppen nicht aus.



Syntax

StopMove()

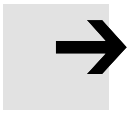
Eine Anwendung für diesen Stopp ist z. B. das Fahren auf ein Hindernis, das mit einem Sensor erkannt wird. Nach dem Erkennen des Zustandes wird über den Befehl **StopMove** die Bewegung gestoppt. In Kapitel 20.1 Stoppen von Bewegungen auf Seite 194 ist ein Beispiel mit Verwendung des **StopMove** Befehls beschrieben.

9.9 Stoppen des Programms ‹StopProgram›

Der Befehl StopProgram hält das Programm an, das Programm geht es den Zustand gestoppt. Er entspricht der Stopp Taste auf dem Handbediengerät. Eine Weiterführung kann nur durch einen erneuten Start z. B. über das Handbediengerät oder von extern über eine SPS Steuerung ausgeführt werden.

Der Befehl ist im Hauptlauf aktiv, dies bedeutet er wird durch den Satzvorlauf nicht ausgeführt. Eine Ausführung der vorherigen Befehle, die durch den Satzvorlauf berechnet werden, ist sichergestellt.

Das Stoppen der Kinematik erfolgt mit der maximalen Bremsrampe, die für das Stoppen der Kinematik definiert ist. Reduktionen der Dynamik durch einen Override wirken sich beim Stoppen nicht aus.



Syntax

StopProgram()

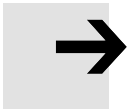
Beispiel:

```
:
Vel(dynCart, 1000)
Lin(pos1)
Lin(pos2)
SetInfo("insert workpiece and press start")
StopProgram()
Lin(pos3)
CALL Conture1
:
```

10. Dynamikbefehle

Mit den Dynamikbefehlen können Geschwindigkeit, Beschleunigung und Ruck für die Bewegungen der Kinematik programmiert werden. Dabei gibt es die Möglichkeit, die Dynamik von Point-To-Point-Bewegungen (PTP) und kartesischen Bewegungen getrennt einzustellen.

Eine Änderung der Dynamikwerte kann an jeder gewünschten Programmzeile erfolgen.



Hinweis

In der Konfiguration (Festo Configuration Tool) werden Initialwerte für die Dynamik angegeben. Diese Werte sind bei einem Programmstart als Initialwert aktiv. Wird im Programm keine Dynamik programmiert, so werden diese Initialwerte verwendet. Mit den folgenden Dynamikbefehlen können diese Werte innerhalb des Programms jederzeit überschrieben werden.

Grafik mit Einstellung der Dynamikwerte im Festo Configuration Tool:

	Standard	Repositionieren	Tippen	Einheit
Geschwindigkeit	100	10	50	%
Beschleunigung	100	10	50	%
Ruck	100	10		%
Geschwindigkeit	4000	100	100	mm/s
Beschleunigung	60000	100	100	mm/s²
Ruck	2000000	200000		mm/s³
Geschwindigkeit	360	360	360	°/s
Beschleunigung	1000	1000	2000000	°/s²
Ruck	2000000	2000000		°/s³

10.1 Autom. Begrenzung der Dynamik, Dynamiklimiter

Um eine Bahntreue einzuhalten, müssen die Solldynamikwerte mit den physikalisch möglichen Dynamikwerten überprüft werden, damit es zu keiner Überschreitung der maximalen möglichen Werte der einzelnen Achsen kommt. Dabei handelt es sich um die Geschwindigkeit, Beschleunigung und den Ruck. Diese maximalen Werte sind in der Konfiguration der einzelnen Achsen hinterlegt.

Die CMXR Steuerung verfügt über einen Begrenzer der Dynamikwerte, der Dynamiklimiter genannt wird. Dieser arbeitet in der Vorausberechnung des FTL Programms und prüft ständig die zu fahrende Dynamikwerte mit der maximal möglichen Dynamik der einzelnen Achsen ab.

Hinweis

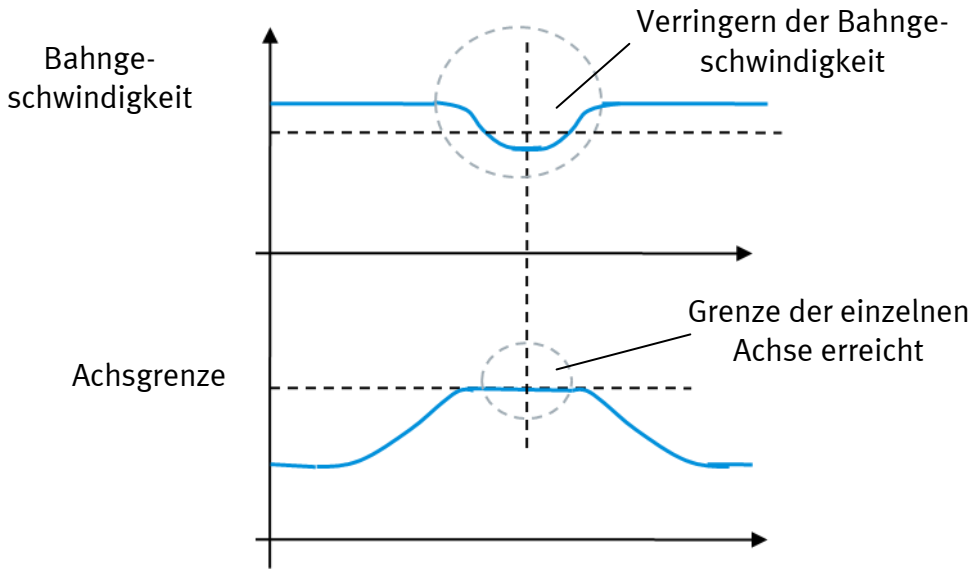
Der Dynamiklimiter ist ständig aktiv und muss nicht aktiviert werden.

Ein Eingreifen des Dynamiklimiters hängt ab von:

- Größe der programmierten Dynamik
- Konstellation der Bewegungsbahn, aus der die Dynamikwerte für die einzelnen Achsen resultieren.

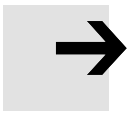
Wird aufgrund der programmierten Dynamik oder das Ändern der Bewegungsbahn das Maximum von mindestens einer Achse überschritten, so wird die Bahndynamik so reduziert, dass die betroffene Achse sich an ihren Grenzen bewegt. Die Bahn wird dabei nicht verlassen. Verlässt die Kinematik kritische Bewegungsbahnen und eine höhere Dynamik ist möglich, so wird dies erkannt und auf den programmierten Wert beschleunigt.

Im folgenden Beispiel wird die Grenze einer Achse erreicht, so dass automatisch die Bahngeschwindigkeit reduziert und somit die Bahntreue gewährleistet wird. Nachdem die Achse den kritischen Bereich verlassen hat, wird wieder auf den programmierten Wert beschleunigt.



10.2 Geschwindigkeiten ‹Vel›

Mit dem Befehl **Vel** kann die Geschwindigkeit für eine kartesische und eine PTP-Bewegung angegeben werden. Die Steuerung reduziert die angegebenen Werte auf die maximal möglichen Achsgeschwindigkeiten der einzelnen beteiligten Achsen. Kommt es zu einer Begrenzung aufgrund der maximalen möglichen Achsgeschwindigkeiten, so wird eine Meldung ausgegeben.



Syntax

Vel (‹Mode› : ENUM, ‹Value› : REAL)

Parameter	Bedeutung	Einheit
Mode	Art der Geschwindigkeit	Enumeration: dynPtp, dynCart
Value	Geschwindigkeitswert	Geschwindigkeitsangabe

Tabelle 10.1 Parameter Befehl Vel

Enumeration Parameter Mode	Bewegungsart	Einheit
dynPtp	Point To Point	%
dynCart	kartesisch	mm/sec

Tabelle 10.2 Einheiten Parameter Value

Beispiel:

Variable:

```
:
axis0 : AXISPOS := (-60, -60, 0, 0, 0, 0, 0, 0, 0)
axis1 : AXISPOS := (60, 60, 0, 0, 0, 0, 0, 0, 0)
axis2 : AXISPOS := (100, 60, 0, 0, 0, 0, 0, 0, 0)
:
```

Programm:

```
:
Vel(dynPtp, 30)           // Geschwindigkeit für PTP auf 30 %
Ptp(axis0)
Vel(dynCart, 500)         // Bahngeschwindigkeit auf 500 mm/sec
Lin(axis1)
speed := 85
```

```
Vel(dynPtp, speed)        // Geschwindigkeit für PTP auf 85 %
Ptp(axis3)
:
```

10.3 Beschleunigung ‹Acc›

Einstellen der Beschleunigung und Verzögerung für PTP- und kartesische Bewegungen der Handachsen. Die nachfolgenden Befehle werden auf den angegebenen Wert reduziert. Die Steuerung reduziert die Beschleunigung bzw. Verzögerung bei Überschreitung der Achsgrenzwerte selbstständig.

Syntax

Acc (‹Mode› : ENUM, ‹ValueAcc› : REAL, OPT ‹ValueDec› : REAL)

Parameter	Bedeutung	Einheit
Mode	Art der Beschleunigung	Enumeration: dynPtp, dynCart
ValueAcc	Wert für die Beschleunigung	mm/sec² oder Grad/sec²
ValueDec	Wert für das Abbremsen, optionale Angabe	mm/sec² oder Grad/sec²

Tabelle 10.3 Parameter Befehl Acc

Enumeration Parameter Mode	Bewegungsart	Einheit
dynPtp	Point To Point	%
dynCart	kartesisch	mm/sec²

Tabelle 10.4 Einheiten Parameter ValueAcc, ValueDec

Hinweis

Wird der optionale Parameter ValueDec (für die Bremsrampe) nicht angegeben, so wird für die Bremsrampe der Wert des Parameters ValueAcc (für die Beschleunigung) verwendet. Das Profil ist dann symmetrisch.

Beispiel:

Variable:

```
:
pos0 : AXISPOS := (-60, -60, 0, 0, 0, 0, 0, 0, 0)
pos1 : AXISPOS := (60, 60, 0, 0, 0, 0, 0, 0, 0)
pos2 : AXISPOS := (100, 60, 0, 0, 0, 0, 0, 0, 0)
```

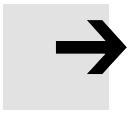
:

Programm:

```
:
Ptp(pos0)
Acc(dynPtp, 30, 30)        // Beschleunigung für PTP auf 30 %
Ptp(pos1)
Acc(dynCart, 100)          // Bahnbeschleunigung auf 100mm/sec²
Lin(pos2)
:
```

10.4 Ruck ‹Jerk›

Einstellen des Rucks für PTP- und kartesischeBewegungen. Die nachfolgenden Befehle werden auf den angegebenen Wert reduziert. Die Steuerung reduziert den Ruck bei Überschreitung der Achsgrenzwerte selbstständig.



Syntax

Jerk (‹Mode› : ENUM, ‹Value› : REAL)

Parameter	Bedeutung	Einheit
Mode	Art der Bewegung	Enumeration: dynPtp, dynCart
Value	Wert des Rucks	

Tabelle 10.5 Parameter Befehl Jerk

Enumeration Parameter Mode	Bewegungsart	Einheit
dynPtp	Point To Point	%
dynCart	kartesisch	mm/sec³

Tabelle 10.6 Einheiten Parameter Value

Beispiel:

Variable:

```
:
pos0 : AXISPOS := (-60, -60, 0, 0, 0 ,0, 0, 0, 0)
pos1 : AXISPOS := (60, 60, 0, 0, 0, 0, 0, 0, 0)
pos2 : AXISPOS := (100, 60, 0, 0, 0, 0, 0, 0, 0)
```

:

Programm:

:

```
Ptp(pos0)
Jerk(dynPtp, 50)          // Ruck für PTP auf 50 %
Ptp(pos1)
Jerk(dynCart, 5000)       // Ruck auf der Bahn auf 5000mm/sec³
Lin(pos2)
```

:

10.5 Override

Mit einem Override lassen sich die Dynamikwerte prozentual einstellen. Dabei lassen sich die Werte für Beschleunigung, Geschwindigkeit und Ruck einfach beeinflussen. Die programmierte Bahn wird nicht beeinflusst.

Hinweis

Soll nur die Geschwindigkeit reduziert werden, so sollte dies über die direkte Angabe des Geschwindigkeitswertes erfolgen. Eine Reduktion über den Override beeinflusst auch die Beschleunigung und den Ruck. Dies bewirkt, dass die Achsdynamik nicht voll ausgenutzt wird und die Bewegung in der Summe langsamer ist.

Es gibt 2 verschiedene Override:

- Dynamik Override, beeinflusst die eingestellten Werte für Geschwindigkeit, Beschleunigung und Ruck.
- Override am Handbediengerät

Wirkungsweise der Override

Dynamik = programmierte Dynamik * Dynamik Override

10.5.1 Override am Handbediengerät ‹Ovr›

Der Override entspricht der Einstellung am Handbediengerät CDSA-D1-VX über die Tasten V+, V-. Der Override wird in der Einheit Prozent angegeben, wobei 100 Prozent der maximalen Dynamik entspricht, die programmiert wurde. Eine Verringerung des Overrides bewirkt ein Verlangsamen der Dynamik, wobei die Bahn dadurch nicht verändert wird.

Die Abbildung zeigt die Tasten V- und V+ zum Einstellen des Override am Handbediengerät CDSA-D1-VX.

Der Override ist sehr hilfreich bei Inbetriebnahmen. Im Automatikbetrieb sollte dieser jedoch auf 100 % stehen, damit die volle Dynamik genützt werden kann. Entsprechende Dynamikanpassungen sind direkt mit den Befehlen für Geschwindigkeit und Beschleunigung vorzunehmen.

Mit dem Befehl **Ovr** kann ein Overridewert direkt im Programm gesetzt werden. Dies hat dieselbe Auswirkung wie eine Änderung über die Tasten am Handbediengerät.

Syntax

Ovr (‹Value› : REAL)

Parameter	Bedeutung	Einheit
Value	Overridewert	Prozent

Tabelle 10.7 Parameter Befehl Ovr

Hinweis

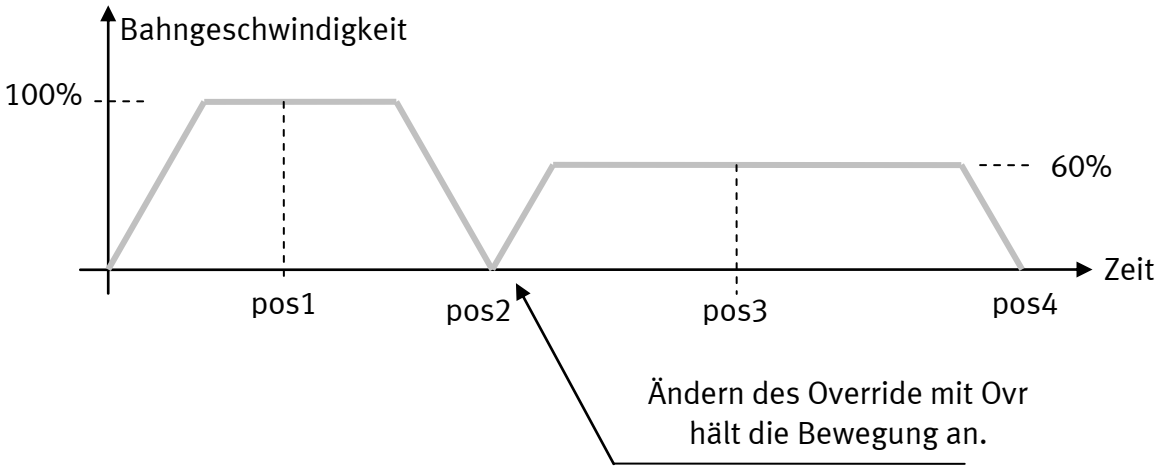
Um die volle Dynamik der Kinematik zu nützen, sollte im Automatikbetrieb der Override stets auf 100 % stehen. Dynamikanpassungen sind über die entsprechenden Befehle direkt vorzunehmen.

Wirkungsweise:

Wird der Override am Handbediengerät geändert, so wirkt sich dieser sofort auf die Bewegung aus. Bedingt durch die Vorausberechnung des Programms kann eine sofortige Änderung des Override über das Makro Ovr nicht ausgeführt werden. Das Verwenden von Ovr hält deshalb die Vorausberechnung an, dies führt zu einen **Stop auf der Bahn.** Ein überschleifen auf das nächste Bahnsegment ist nicht möglich.

Beispiel:

```
// setzen des Override auf 100%
Ovr(100)
Lin(pos1)
Lin(pos2)
// setzen des Override auf 60%
Ovr(60)
Lin(pos3)
Lin(pos4)
```



10.5.2 Dynamikoverride ‹DynOvr›

Mit dem Befehl **DynOvr** werden die eingestellten oder programmierten Dynamikwerte beeinflusst. Er unterliegt dem Override, welcher am Handbediengerät eingestellt wird.

Syntax

DynOvr (‹Value› : REAL)

Parameter	Bedeutung	Einheit
Value	Wert des dynamischen Overrides	Prozent

Tabelle 10.8 Parameter Befehl DynOvr

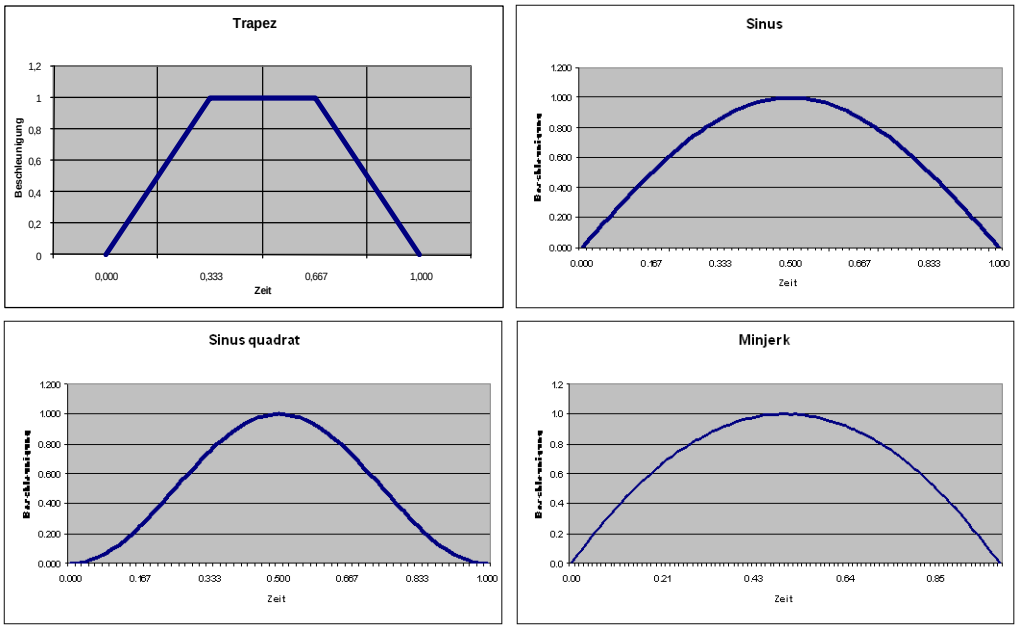
Hinweis

Die gefahrene Bahn wird durch den Override nicht verändert. Der programmierte Wert hat keine Auswirkungen auf die Vorausberechnung des Programmes.

10.6 Beschleunigungsrampen

Mit dem Befehl **Ramp** kann eine Rampenform zur Beschleunigung bzw. Verzögerung eingestellt werden. Zur Auswahl stehen vier Rampenformen, wobei nach dem Start die Sinus-Rampe aktiv ist.

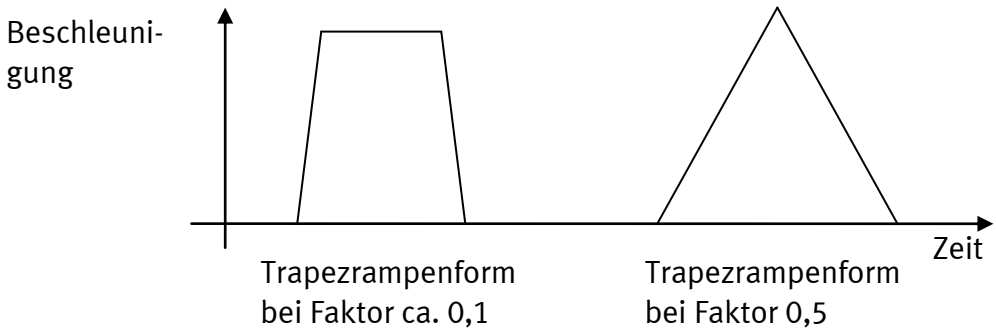
Abbildung der 4 Rampenformen:



Trapez-Rampe

Die Trapez-Rampe bewirkt einen trapezförmigen Beschleunigungsverlauf. Der Ruck verläuft somit rechteckförmig. Mit der Trapez-Rampe können die schnellsten Ausführungszeiten der Bahn realisiert werden.

Mit einem optionalen Parameter lässt sich die Rampenform beeinflussen. Dieser darf den Wert › 0 und ‹= 0,5 besitzen. Bei einem Wert von 0,5 ergibt sich ein Beschleunigungsdreieck, bei einem Wert z. B. 0,1 nähert sich die Rampenform einem Rechteckprofil.



Wird der optionale Parameter nicht angegeben, wird er automatisch auf den Faktor 0.5 gestellt. Daraus ergibt sich das Beschleunigungsdreieck.

Sinus-Rampe

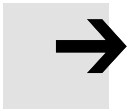
Die Sinus-Rampe hat den Vorteil, dass nicht nur die Beschleunigung, sondern auch der Ruck sinusförmig verläuft. Dies bewirkt einen weicheren Beschleunigungsverlauf als die Trapez-Rampe. Jedoch ist die Ausführungszeit zur Geschwindigkeitserreichung etwas höher.

Sinus-Quadrat-Rampe

Die Sinus-Quadrat-Rampe hat den weichsten Bahnverlauf, bewirkt aber auch die längste Ausführungszeit beim Fahren der Bahn.

MinimumJerk-Rampe

Die Minimumjerk-Rampe ist eine spezielle Rampenform, die einen Kompromiss zwischen Ausführungszeit und Weichheit der Bewegung darstellt. Der Beschleunigungsverlauf ähnelt einem Sinusprofil, der Ruckverlauf ist jedoch nicht sinusförmig sondern sägezahnartig. Dies bewirkt eine weiche Bewegung bei doch schneller Ausführungszeit.

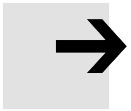


Hinweis

Die Verwendung eines Rampentyps hängt von der eingesetzten Kinematik und von der Applikation ab. Nach einer Auswahl der Rampenform ist diese in der Bewegung zu testen.

10.6.1 Setzen von Rampenformen ‹Ramp›

Mit dem Befehl **Ramp** kann eine Rampenform ausgewählt werden. Diese wird für den Bewegungsverlauf aller folgenden Bewegungsbefehle verwendet.



Syntax

Ramp(‹Ramptype› : ENUM, OPT ‹Param› : REAL)

Parameter	Bedeutung	Einheit
Ramptype	Rampentyp, wählt die Rampenform aus.	Enumeration: TRAPEZOID SINE SINESQUARE MINJERK
Param	Parameter für Trapezrampen	-----

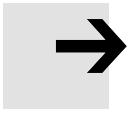
Tabelle 10.9 Parameter Befehl Ramp

Beispiel:

```
:
Ramp(TRAPEZOID)      // Auswahl Trapezrampe
Lin(pos1)
Lin(pos2)
WaitTime(1000)
Ramp(SINE)           // Auswahl der Sinusrampe
Lin(pos3)
Lin(pos4)
:
```

10.7 Konstante Bahngeschwindigkeit einschalten ‹VconstOn›

Mit diesem Befehl wird die Überwachung der konstanten Bahngeschwindigkeit eingeschaltet. Diese wirkt sich nur auf kartesische Befehle wie z. B. **LIN** und **CIRC** aus. Bei PTP-Bewegungen hat dieser Befehl keine Auswirkungen.

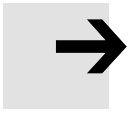


Syntax

VconstOn (‹Tolerance› : REAL, ‹StopOnViolation› : BOOL)

Parameter	Bedeutung	Einheit
Tolerance	Prozentualer Wert für den zulässigen Einbruch der Bahngeschwindigkeit	Prozentwert 0 % bis 100 %
StopOnViolation	Bei TRUE Auslösen eines Fehlers bei Verletzung der Toleranz.	Schalter: TRUE oder FALSE

Tabelle 10.10 Parameter Befehl VconstOn



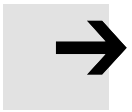
Hinweis

Wird eine konstante Bahngeschwindigkeit benötigt, so ist darauf zu achten, dass für den Überschleifbereich ein geometrisches Überschleifen eingestellt ist (siehe Kapitel 11.4 Geometrisches Überschleifen auf Seite 107). Ein Überschleifen auf Basis der prozentualen Geschwindigkeit führt zu Veränderung der Bahngeschwindigkeit im Überschleifbereich.

Die CMXR Mehrachssteuerung berechnet anhand der Bahn und den maximalen Dynamikwerten der Mechanik die mögliche Bahngeschwindigkeit. Die Grenze der möglichen Bahngeschwindigkeit wird durch die Dynamik der Mechanik bestimmt. Falls Bahnsegmente

programmiert werden, die bedingt durch die Dynamikgrenzen nicht mit konstanter Bahngeschwindigkeit gefahren werden können, kommt es an diesen Stellen zu Einbrüchen der Bahngeschwindigkeit.

Mit dem Parameter **Tolerance** kann ein prozentualer Wert für den zulässigen Einbruch der Bahngeschwindigkeit angegeben werden. Wird als Toleranzwert 100 % angegeben, so ist die Überwachung ausgeschaltet.



Hinweis

Die Überwachung der Bahngeschwindigkeit berücksichtigt nicht den eingestellten Override am Handbediengerät. D.h. wenn die Kinematik aufgrund eines begrenzenden Overrides nicht ihre volle Geschwindigkeit erreicht, wird die Überwachung aktiv und ein Fehler ausgelöst.

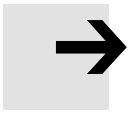
Beispiel:

Programm:

```
:
Lin(pos1)
VconstOn(25, TRUE)
Lin(pos2)
Lin(pos3)
VconstOff()
:
```

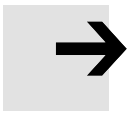
10.8 Konstante Bahngeschwindigkeit ausschalten ‹VconstOff›

Mit diesem Befehl wird eine konstante Bahngeschwindigkeit ausgeschaltet.



Syntax

VconstOff ( )



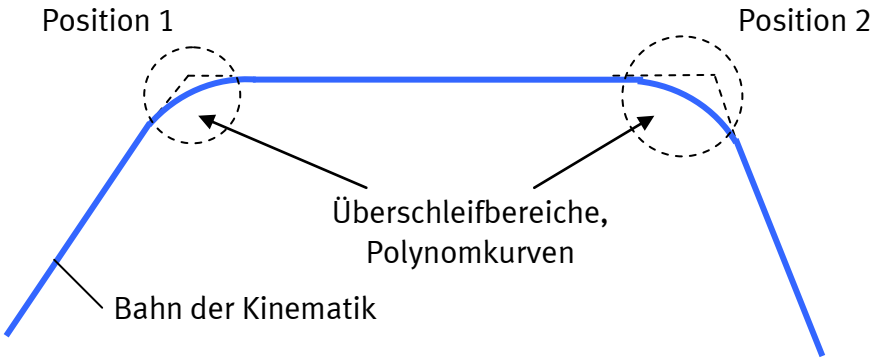
Hinweis

Ein Abbruch des Anwenderprogramms führ nicht automatisch zum Abschalten der Überwachung der Bahngeschwindigkeit. Bei einem Neustart des Anwenderprogramms muss zunächst immer die Überwachung abgeschaltet werden.

11. Überschleifbefehle

Unter Überschleifen versteht man das Beschleunigen der Achsen zum Anfahren der nächsten Position, obwohl die vorherige Position noch nicht erreicht wurde. Oft ist es nicht notwendig, dass eine Position genau erreicht wird, sondern dass die Bewegung schnell und schonend, d. h. mit möglichst geringer Belastung für die Mechanik angefahren wird. Hier bietet die Überschleiffunktion die Möglichkeit, die Genauigkeit zum Erreichen eines Punktes sowie die Härte der Bewegung einzustellen.

In der folgenden Abbildung ist die Funktionsweise der Überschleiffunktion dargestellt.



Das Bewegungsprogramm enthält eine Positionierung auf Position 1, dann auf Position 2. Durch die Überschleiffunktion, die über einen Programmbefehl eingestellt wird, werden die einzelnen Positionen nicht genau angefahren. Die Dynamikprofile zum Anfahren der einzelnen Positionen werden in den Überschleifbereichen überlagert, was zu einer Erhöhung der Dynamik führt.

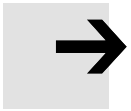
Hinweis

Nach dem Laden eines Programms ist kein Überschleifen aktiv, d. h. es erfolgt ein genaues Anfahren der Positionen. Ein gewünschtes Überschleifen muss mit den entsprechenden Funktionen aktiviert werden.

Das Überschleifen kann auf zwei Arten erfolgen:

1. Geschwindigkeitsüberschleifen anhand der Geschwindigkeit
2. Positionsüberschleifen anhand einer vorgegebenen Distanz

Die Beschreibung dieser Arten erfolgt auf den folgenden Seiten.



Hinweis

Die Bahn im Überschleifbereich wird durch eine Polynomkurve bestimmt, die sich aus mathematischen Berechnungen ergibt. Diese verwirklicht einen stetigen Anstieg der Dynamik und somit eine schonendes Verhalten für die Mechanik. Mit einem Verrunden über einen Radius kann dieses Verhalten nicht erzielt werden. Somit kann eine Verrundung von Bahnsegmenten mit dem Überschleifen nicht realisiert werden.

11.1 Nullsegmente

Wird ein Nullsegment programmiert, d. h. die erneute Positionierung auf die bereits angefahrene Position , so kann nicht überschliffen werden. Dies führt zu einem Abbremsen mit anschließendem Beschleunigen auf der Bahn.

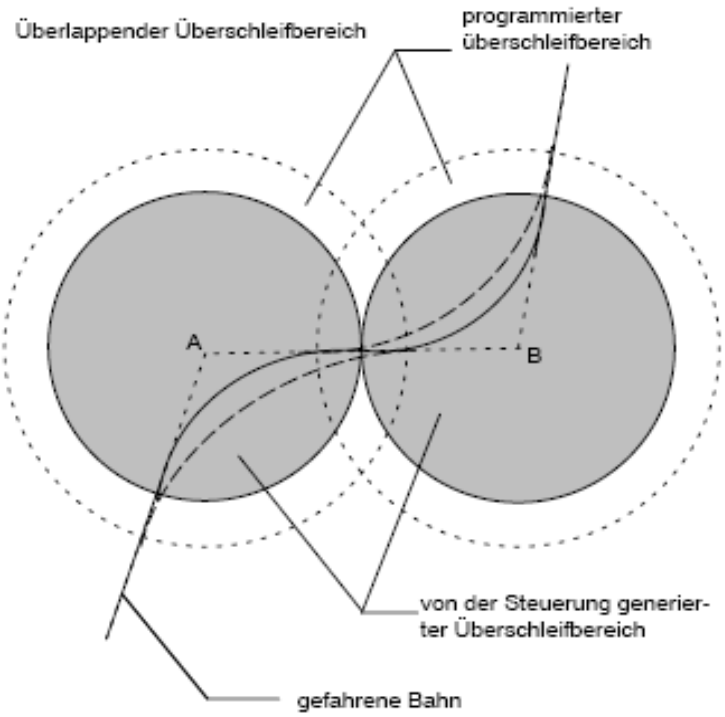
Beispiel:

```
Lin(pos1)
Lin(pos2)
// erneute Positionierung auf pos2 = kein Überschleifen möglich
Lin(pos2)
Lin(pos3)
```

Da die Position pos2 wiederholt programmiert ist, kommt es an dieser Programmstelle zu einem Stopp auf der Bahn.

11.2 Grenzbereich

Ist ein Abstand zwischen 2 Punkten kleiner als notwendig, um dazwischen die parametrierte Überschleifbahn zu fahren, reduziert die Steuerung den Überschleifbereich zwischen den Punkten automatisch, so dass der bestmögliche Überschleifwert gefahren werden kann.



In der Abbildung sind die gestrichelten Kreise der Überschleifbereich, der sich aufgrund der Parametrierung ergibt. Die Kreise überlappen sich, da die Distanz zwischen der Position A und B nicht ausreicht um dieses Profil abzufahren. Die Steuerung berechnet nun automatisch den maximal möglichen Überschleifbereich, der durch die grauen Vollkreise dargestellt ist.

Hinweis

Das Überschleifen ist auf 50 % der Bahnsegmentlänge beschränkt. Ist der Überschleifbereich größer als maximal möglich, verkürzt die CMXR Mehrachssteuerung den Überschleifbereich automatisch auf 50 % der Bahnsegmentlänge.

Hinweis

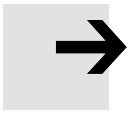
Zu kleine Bahnsegmentlängen können zu ungewollten Dynamikeinbrüchen führen, falls der definierte Überschleifbereich reduziert wurde. Falls dies ungewünscht ist, müssen die Bahn oder der Überschleifbereich angepasst werden.

11.3 Geschwindigkeitsüberschleifen

Beim Geschwindigkeitsüberschleifen werden die Dynamikprofile der Bahn überlappt. Daraus resultiert eine Bewegung im Überschleifbereich auf die nächste Position.

11.3.1 Mit Prozentfaktor ‹OvlVel›

Beim Geschwindigkeitsüberschleifen wird ein Grad des Überschleifens mit einem Prozentwert vorgegeben. Der Wertebereich beträgt 0 % bis 200 %.



Syntax

OvlVel (‹Value› : REAL)

Parameter	Bedeutung	Einheit
Value	Überschleifwert	Prozent

Tabelle 11.1 Parameter Befehl OvlVel

Parameter:

ProzentÜberschleifparameter in %

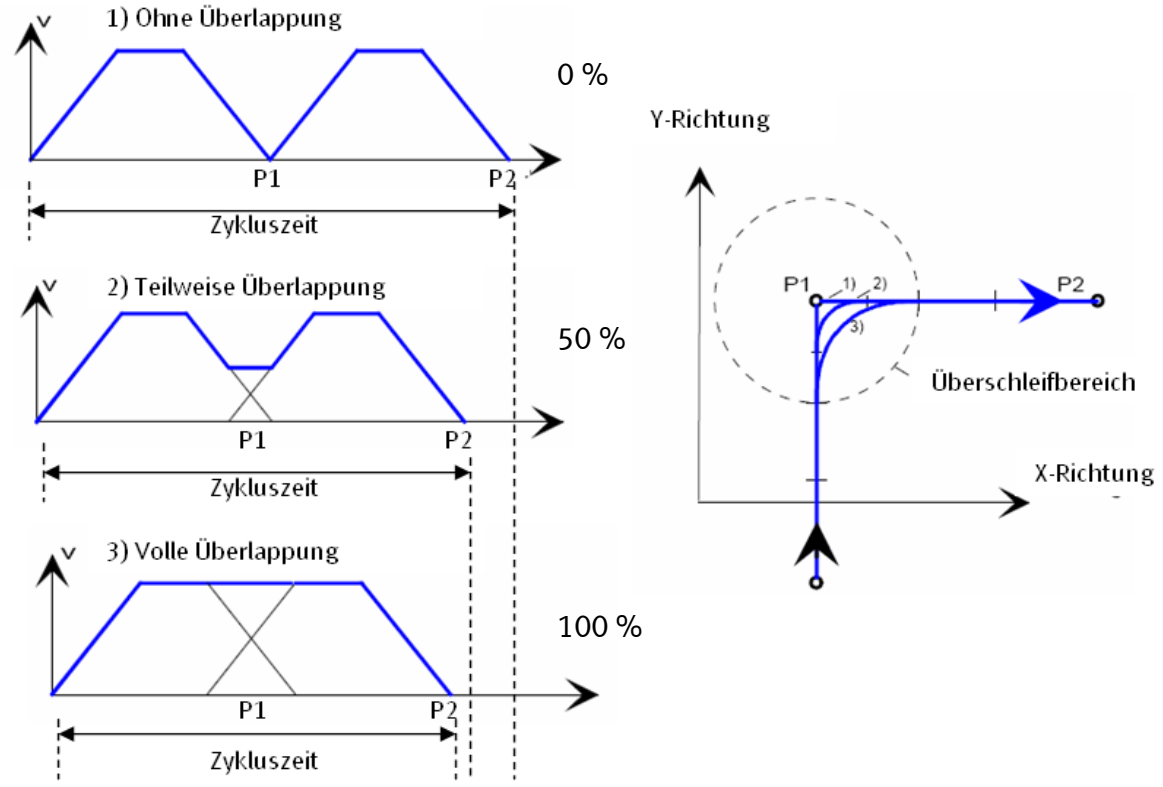
0 % ohne Überschleifen

100 % optimale Nutzung der Achsbeschleunigungen

100..200 % ohne Zeitverlust, weichere Bewegung mit größerem Überschleifbereich

Werte kleiner als 100 % bewirken kleinere Abweichungen zur Position, verlängern jedoch die Bewegungszeit, da die Geschwindigkeit reduziert werden muss. Bei einem Wert von 100 % werden alle Beschleunigungsreserven der Achsen ausgenützt, wobei eine möglichst geringe Abweichung von der Position berücksichtigt wird. Gibt man Werte zwischen 100 % und 200 % an, so werden die Positionabweichungen (Schleppfehler) erhöht, wobei die Achsbeschleunigungen im Gegensatz zu einer Einstellung von 100 % verringert werden.

Folgende Abbildung zeigt die Geschwindigkeitsprofile bei einer Fahrt auf Position 1 und Position 2. Dabei sind verschiedene Werte für den Überschleifbereich definiert.



Grafik 1 zeigt ein Geschwindigkeitsprofil, bei dem keine Überlappung stattgefunden hat. Die Achsen bremsen auf der Bahn ab; somit wird die Position 1 und 2 genau erreicht. In der mittleren Grafik ist eine teilweise Überlappung dargestellt, in der unteren Grafik eine volle Überlappung (100 %) der Geschwindigkeitsprofile.

Hinweis

Im Überschleifbereich wird mit der vollen Achsdynamik gearbeitet. Dies bedeutet, dass das Bahnstück im Überschleifbereich kein Radius ist, sondern eine Polynomkurve, die sich aus den aktuellen Dynamikwerten der Achsen ableitet.

Beispiel:

```
:
OvlVel(100)      // Überschleifen mit 100 %
Lin(pos1)
Lin(pos2)
OvlVel(75)       // Überschleifen mit 75 %
Lin(pos3)
:
```

11.4 Geometrisches Überschleifen

Beim geometrischen Überschleifen werden die Abweichungen von der programmierten Endposition festgelegt. Im Gegensatz zum Geschwindigkeitsüberschleifen wird der Versatz geometrisch d. h. in Längen- oder Winkeleinheiten angegeben.

Beim geometrischen Überschleifen wird zwischen einem Überschleifen der kartesischen Achsen X, Y und Z sowie der Orientierungsachsen unterschieden.

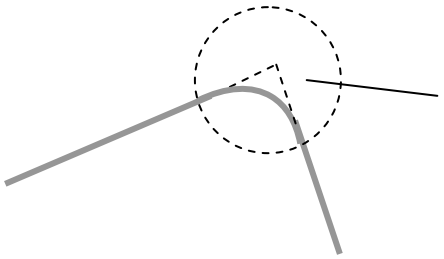
Hinweis

Das geometrische Überschleifen kann nur bei kartesischen Bewegungen benutzt werden. PTP-Bewegungen sind mit dieser Art des Überschleifens nicht möglich.

Es eignet sich nicht zum Verrunden von Ecken, da die geometrische Form im Überschleifbereich kein Radius, sondern eine Polynomkurve ist. Ein Radius ist zum Überschleifen ungeeignet, da dies einen sprunghaften Anstieg der Beschleunigung zur Folge hat.

11.4.1 Überschleifen der Achsen X, Y und Z ‹OvlCart›

Das Überschleifen von kartesischen Bewegungen mit einer geometrischen Bestimmung wird durch die Angabe eines Abstandes des TCP auf der Bahn bis zum Zielpunkt angegeben.



Die Überschleifbewegung wird beim Eintritt in die Kugel gestartet und an dem Punkt, an dem die Kugel das nächste Bahnstück schneidet, beendet. Diese Kurve verläuft tangential zu den 2 betroffenen Bahnstücken. Die Grenze des Überschleifbereiches bildet die Hälfte des kürzesten der beteiligten Bahnsegmente.

Die Bahn im Überschleifbereich ist kein Radius, sondern ein Polynom 5. Ordnung. Dieses Polynom erzeugt den maximal möglichen weichen Bahnverlauf, was mit einem Kreisbogen nicht möglich ist.

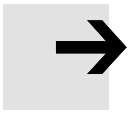
Syntax

OvlCart (‹Distance› : REAL)

Parameter	Bedeutung	Einheit

Parameter	Bedeutung	Einheit
Distance	Überschleifbereich, Abstand zum Endpunkt	eingestellte Längeneinheit

Tabelle 11.2 Parameter Befehl OvlCart

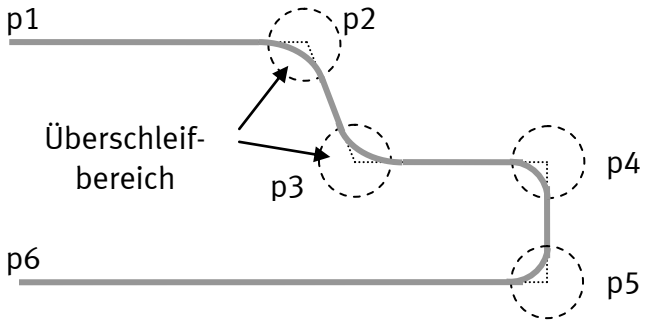


Hinweis

Ein geometrisches Überschleifen wird sehr oft in Verbindung mit einer konstanten Bahngeschwindigkeit benötigt. Diese wird mit dem Befehl **VconstOn** eingestellt (siehe Kapitel 10.7 Konstante Bahngeschwindigkeit auf Seite 99)

Beispiel:

Eine Kontur soll mit konstanter Bahngeschwindigkeit und mit einem Überschleifbereich von 5 mm abgefahren werden.




```
Vel(dynCart, 300)     // Bahngeschwindigkeit auf 300 mm/sec
VconstOn(25, TRUE)    // einschalten konst. Bahngeschw.
OvlCart(5)            // Überschleifbereich einstellen
Lin(p1)
Lin(p2)
Lin(p3)
Lin(p4)
Lin(p5)
Lin(p6)
```

12. Referenzsysteme (Nullpunktverschiebung)

Referenzsysteme sind kartesische Koordinatensysteme mit 3 translatorischen und 3 rotatorischen Freiheitsgraden. Die Definition der Orientierung basiert auf der Euler ZYZ Konvention.

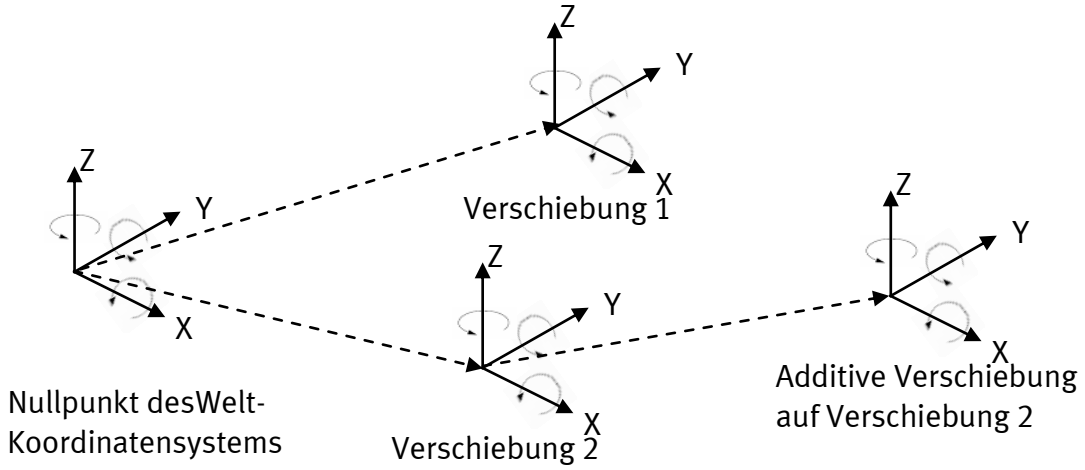
Hinweis

Da die Referenzsysteme von kartesischer Art sind, haben sie keine Auswirkungen auf den Nullpunkt einzelner Achsen im Achskoordinatensystem. Sie wirken sich nur auf das kartesische Koordinatensystem aus.

Zum Start eines Programms ist das Welt-Koordinatensystem aktiv.

12.1 Bezug des Referenzsystems

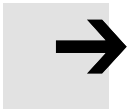
Ein Referenzsystem bezieht sich auf den kartesischen Nullpunkt eines schon definierten Koordinatensystems. Die definierten Werte des neuen Referenzsystems bewirken eine Verschiebung in den 6 Freiheitsgraden.



Wie in der Abbildung ersichtlich ist, lassen sich mehrere Referenzsysteme auf den kartesischen Nullpunkt eines anderen Referenzsystems definieren, wobei nur eine Referenz aktiv sein kann.

Hinweis

Ein Verschachteln von Referenzsystemen ist mit Bedacht anzuwenden. Dieser Mechanismus kann unter Umständen zu einer effizienten Programmierung dienen. Jedoch ist die Transparenz beim Lesen des Programms schwierig und nicht bedachte Verschachtelungen können evtl. zu Kollisionen führen.



Hinweis

Die Verschiebung des Referenzsystems setzt sich aus einer Translation und einer Verdrehung (Orientierung) zusammen. Bei der Ausführung wird zuerst die Verschiebung, danach die Verdrehung ausgeführt.

12.2 Daten des Referenzsystems

Die Daten eines Referenzsystems bestehen aus einer 3-dimensionalen Translation und einer 3-dimensionalen Orientierungsangabe. Die Definition der Orientierung erfolgt nach der Euler-ZYZ-Methode.

Diese Daten werden in einer Variable angelegt, die aus einem strukturierten Datentyp besteht. Der Name des Referenzsystems ist frei wählbar. Die Anzahl der Referenzsysteme ist durch den Speicher begrenzt.

Es gibt folgende Möglichkeiten, die Daten für ein Referenzsystem zu definieren:

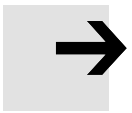
1. direkte Angabe der Werte
2. Angabe über 3 kartesische Punkte

Neben den Werten des Referenzsystems gibt es die Möglichkeit, einen Bezug auf ein anderes Referenzsystem herzustellen. In jedem der strukturierten Datentypen gibt es dazu die Möglichkeit, unter dem Parameter **RefSys** einen Bezug zu einem anderen Referenzsystem herzustellen. Unter dem Parameter **RefSys** können die Datentypen aller Möglichkeiten angegeben werden.

Im Folgenden werden die Befehle zum Setzen eines Referenzsystems beschrieben.

12.3 Referenzsystem mit direkten Werten ‹SetRefSys›

Der Befehl **SetRefSys** aktiviert ein Referenzsystem, dessen Daten absolut in die Datenstruktur der übergebenen Variable eingetragen sind.



Syntax

SetRefSys(‹refSys› : REFSYSDATA)

Parameter	Bedeutung	Einheit
refSys	Referenzsystem, das durch Verschiebungswerte definiert ist.	Längen und Winkeleinheiten

Tabelle 12.1 Parameter Befehl SetRefSys

Bei einer direkten Wertangabe werden die Werte direkt mit der übergebenen Variable bekannt gemacht. Ein Verändern bereits übergebener Daten ist nur über einen erneuten Aufruf möglich.

Aufbau Datentyp REFSYSDATA:

baseRs	: REFSYS	Bezug auf ein anderes Referenzsystem
x	: REAL	Verschiebung entlang der X-Achse
y	: REAL	Verschiebung entlang der Y-Achse
z	: REAL	Verschiebung entlang der Z-Achse
a	: REAL	Orientierung nach Euler, Drehung um die Z-Achse
b	: REAL	Orientierung nach Euler, Drehung um die verdrehte Y-Achse
c	: REAL	Orientierung nach Euler, Drehung um die verdrehte Z-Achse

Unter dem Parameter baseRs kann man eine andere Referenz angeben, auf das sich das Referenzsystem additiv wirkt. Wenn ein Bezug zum Weltkoordinatensystem der Kinematik erfolgen soll, muss auf die Systemvariable **_system.world** referenziert werden.

Beispiel:

Daten:

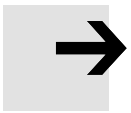
```
refsysdata0 : REFSYSDATA := (MAPX("_system.world"),
              100, 150, 0, 0, 0, 0)
```

Programm:

```
SetRefSys(refsysdata0)
```

12.4 Referenzsystem mit 3 Punkten ‹SetRefSys3P›

Mit dem folgenden Befehl **SetRefSys3P** wird ein Referenzsystem gesetzt, dessen Daten über 3 Positionen im Raum bestimmt werden.



Syntax

SetRefSys3P(‹refSys› : REFSYS3P)

Parameter	Bedeutung	Einheit
refSys	Referenzsystem, das durch 3 Positionen bestimmt ist	Längen und Winkeleinheiten

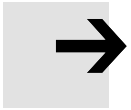
Tabelle 12.2 Parameter Befehl SetRefSys3P

Verwendung:

Diese Art der Beschreibung eines Referenzsystems bietet die Möglichkeit, über 3 Positionen geteacht zu werden. Diese 3 Positionen sind von kartesischer Art mit 6 Freiheitsgraden.

Bedeutung der Positionen:

- Die erste Position bestimmt den Ursprung des Referenzsystems.
- Die zweite Position bestimmt einen Punkt, durch den die positive kartesische X-Achse des Referenzsystems läuft.
- Die dritte Position bestimmt einen Punkt auf der XY-Ebene.



Hinweis

Die Orientierungen der Positionen sind nicht relevant. Für die Berechnungen werden nur die kartesischen Positionen benötigt.

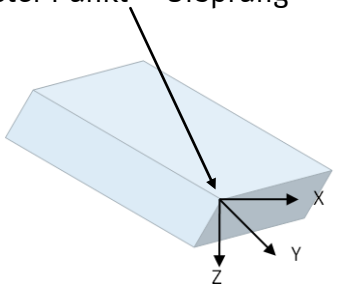
Aufbau Datentyp REFSYS3P:

baseRs	: REFSYS	Bezug auf ein anderes Referenzsystem
p0	: CARTPOS	Ursprung des zu definierenden Referenzsystems
px	: CARTPOS	Position entlang der X-Achse
pxy	: CARTPOS	Position auf der X-Y Ebene

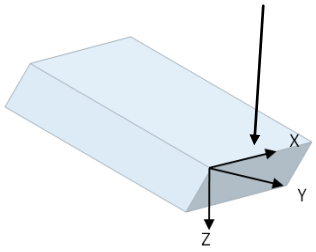
Unter dem Parameter **baseRs** kann man eine andere Referenz angeben, auf die das Referenzsystem additiv wirkt. Wenn ein Bezug zum Weltkoordinatensystem der Kinematik erfolgen soll, muss auf die Systemvariable **_system.world** referenziert werden.

Vorgehensweise beim Teachen:

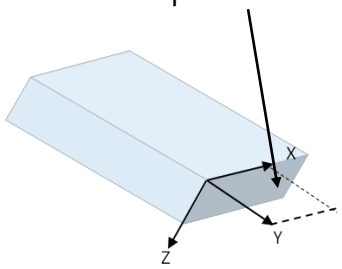
Erster Punkt = Ursprung



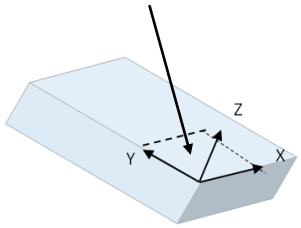
zweiter Punkt = positive X-Achse



dritter Punkt = positive XY-Ebene



dritter Punkt = positive XY-Ebene



Vorsicht

Der dritte Punkt bestimmt die positive XY Ebene. Je nach Lage des 3. Punkte kann dies ein Drehen des Koordinatensystems zur Folge haben z. B. drehen der Z-Achse um 180 Grad.

Beispiel:

Daten:

```
refsys3p0 : REFSYS3P := (MAPX("_system.world"),
                        (100, 100, 0, 0, 0, 0, 0, 0, 0),
                        (200, 100, 0, 0, 0, 0, 0, 0, 0),
                        (200, 200, 0, 0, 0, 0, 0, 0, 0))
```

Programm:

```
SetRefSys3P(refsys3p0)
```

12.5 Referenzsystem Welt ‹SetRefSysWorld›

Mit diesem Befehl wird das Referenzsystem Welt aktiviert, dessen Ursprung in der Kinematikkonfiguration hinterlegt ist.

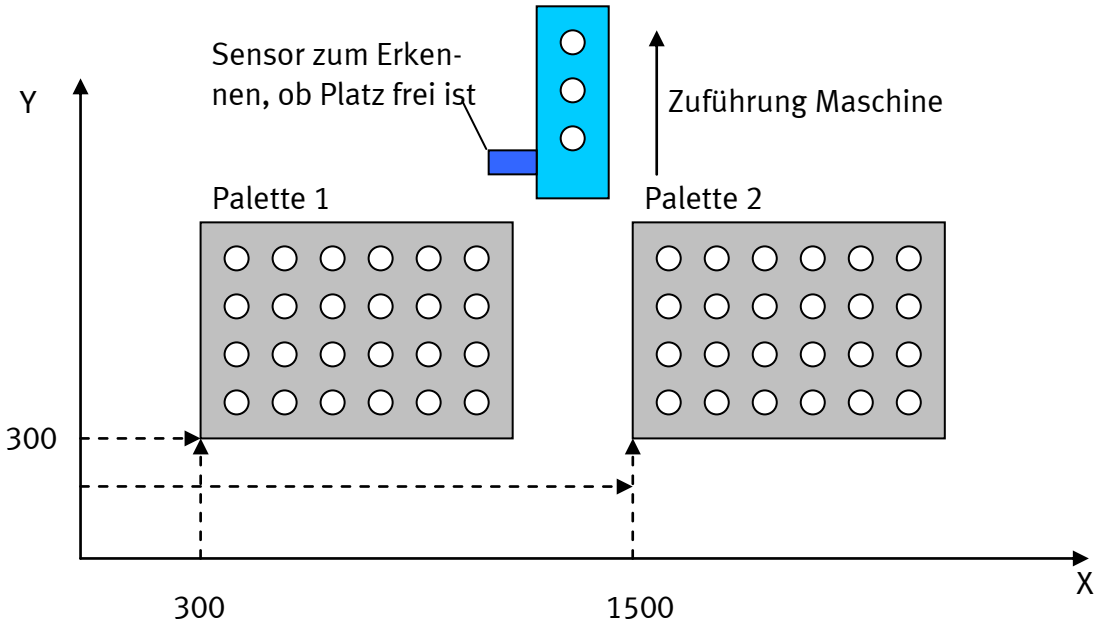
Syntax

SetRefSysWorld()

Wurde ein Referenzsystem mit dem Befehl **SetRefSys** oder **SetRefsys3P** gesetzt d. h. aktiviert und man möchte an bestimmter Programmstelle diese Referenzsysteme deaktivieren, so wird dafür der Befehl **SetRefSysWorld** verwendet.

12.6 Beispiel

Im folgenden Beispiel müssen 2 Paletten entleert und die Teile einer Maschine zugeführt werden.



Beide Paletten besitzen den gleichen Inhalt und die gleichen Abmaße. Um hier den Programmieraufwand gering zu halten, wird für jede Palette ein eigenes Referenzsystem gesetzt und das Programm der Palette in einem Unterprogramm verfasst.

Für das Referenzsystem der Palette 1 wird die Variable RefPal1 und für Palette 2 RefPal2 verwendet.

Daten:

```
:
refPal1 : REFSYSDATA := (MAPX("_system.world"),
            0, 0, 0, 0, 0, 0)
```

```
refPal2 : REFSYSDATA := (MAPX("_system.world"),
            0, 0, 0, 0, 0, 0)
pos1 : CARTPOS := (100, 80, 70, 0, 0, 0, 0, 0, 0)
:
```

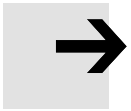
Programm:

```
:
refPal1.x := 300     // Verschiebung Palette 1 in X
refPal1.y := 300     // Verschiebung Palette 1 in Y
refPal2.x := 1500    // Verschiebung Palette 2 in X
refPal2.y := 300     // Verschiebung Palette 2 in Y
Lin(pos1)            // fahren auf Sicherheit im Welt System
SetRefSys(refPal1)   // Verschiebung Palette 1 aktivieren
CALL Feed()          // Aufruf Unterprogramm Teile zuführen
SetRefSys(refPal2)   // Verschiebung Palette 2 aktivieren
CALL Feed()          // Aufruf Unterprogramm Teile zuführen
SetRefSysWorld()     // aktiviere Welt System
Lin(pos1)
:
```

13. Referenzieren einer Kinematik

13.1 Referenzfahrt ‹RefAxis›

Der Befehl **RefAxis** ermöglicht das Referenzieren von Achsen. Die Referenzfahrt wird dabei immer mit einer einzelnen Achse ausgeführt.



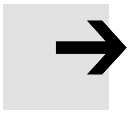
Syntax

RefAxis(‹axis›:AXIS, OPT ‹refData›:REFDATA, OPT ‹timeout›:REAL)

Parameter	Bedeutung	Einheit
Axis	Ausgewählte Achse, die referenziert wird.	Enumeration A1, A2, bis A9
refData	Datensatz der Referenzfahrt, z. B. wie referenziert wird.	keine
timeout	Zeit für den Ablauf der Operation (danach kommt eine Fehlermeldung, die Referenzierung wird abgebrochen).	Sekunden

Tabelle 13.1 Parameter Befehl RefAxis

Bei der Durchführung der Referenzfahrt ist darauf zu achten, dass die beteiligten Achsen freie Fahrt in ihren Bewegungen haben. Die Bewegung wird als Achsbewegung ausgeführt. Je nach Kinematiktyp kann es aufgrund der Achsanordnungen am Werkzeug zu unbekannten Bewegungen kommen.



Hinweis

Die Parameter refData und timeout sind optional. Werden diese nicht angegeben, wird die aktuelle Ist-Position als Referenzposition übernommen (DS 402 - Methode 35).

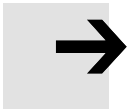
Vorsicht

Bei der Referenzfahrt ist auf freie Fahrt der betroffenen Achsen zu achten. Ferner sind bei diesen Bewegungen geeignete Dynamikwerte zu wählen, um die Referenzfahrt sinnvoll auszuführen. Hohe dynamische Werte sind hier nicht sinnvoll.

Das Referenzieren benötigt zum Ablauf einige Daten. Diese werden in einen Referenzdatensatz vom Typ REFDATA angegeben.

Parameter	Typ	Bedeutung
method	DINT	Referenzfahrmethode nach CANOpen DS 402
offset	REAL	Offset der Referenzposition [mm]
velSwitch	REAL	Referenziergeschwindigkeit (Anfahren des Schalters)
velZero	REAL	Kriechgeschwindigkeit (Suchen der Flanke)
acc	REAL	Referenzfahrbeschleunigung

Tabelle 13.2 Aufbau Datentyp REFDATA



Hinweis

Wird die Referenzfahrtmethode 99 angegeben, werden die Parameter der Referenzfahrt, wie sie mit dem Festo Configuration Tool (FCT) im Antrieb abgelegt wurden, verwendet. Die Referenzfahrt wird dann so ausgeführt, wie bei Inbetriebnahme der einzelnen Achse. Alle andere Parameter wie offset, velSwitch, zeroSwitch und acc haben in diesem Fall keine Bedeutung.

Referenzfahrtmethode

Die Methode der Referenzfahrt kann auf verschiedene Art geschehen, z. B. negativer Endschalter mit Nullimpulsauswertung, positiver Endschalter mit Nullimpulsauswertung oder Referenzfahrt auf einen Endschalter. Diese Methoden sind alle in der zugehörigen CANopen-Dokumentation des jeweiligen Antriebsreglers hinterlegt.

Die folgende Tabelle zeigt die Referenziermethoden für CANopen-Geräte nach DS 402.

Wert	Richtung	Ziel	Bezugspunkt für Null
-18	positiv	Anschlag	Anschlag
-17	negativ	Anschlag	Anschlag
-2	positiv	Anschlag	Nullimpuls
-1	negativ	Anschlag	Nullimpuls
1	negativ	Endschalter	Nullimpuls
2	positiv	Endschalter	Nullimpuls
7	positiv	Referenzschalter	Nullimpuls
11	negativ	Referenzschalter	Nullimpuls
17	negativ	Endschalter	Endschalter
18	positiv	Endschalter	Endschalter
23	positiv	Referenzschalter	Referenzschalter

Wert	Richtung	Ziel	Bezugspunkt für Null
27	negativ	Referenzschalter	Referenzschalter
33	negativ	Nullimpuls	Nullimpuls
34	positiv	Nullimpuls	Nullimpuls
35	-	Keine Fahrt	Aktuelle Ist-Position
99	-	-	Ablauf wie in FCT Projekt der Achse definiert

Tabelle 13.3 Referenzfahrtmethoden

Verschiebung der Referenzposition

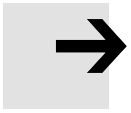
Über den Parameter offset kann eine Verschiebung des Nullpunktes gegenüber der Referenzposition festgelegt werden. Nach dem Referenzieren wird dieser Offsetwert auf den Referenznullpunkt addiert. Die Istwertanzeige der betroffenen Achse wird entsprechend aktualisiert.

Referenziergeschwindigkeit, Kriechgeschwindigkeit, Beschleunigung

Mit der Referenziergeschwindigkeit und der Beschleunigung wird die Dynamik der Achse beim Referenzieren festgelegt. Diese ist wirksam, vom Start der Referenzfahrt, bis zum Erreichen der Flanke des entsprechenden Schalters. Ist die Flanke erkannt, so wird auf die Kriechgeschwindigkeit umgeschaltet und die Referenzfahrt gemäß der ausgewählten Methode fertig gestellt.

13.2 Asynchrone Referenzfahrt ‹RefAxisAsync›

Mit Hilfe dieses Befehls ist es möglich, mehrere Roboterachsen parallel zu referenzieren. Der Befehl wartet dabei nicht bis die Referenzfahrt abgeschlossen ist, sondern der Programmablauf wird nach Absetzen des Referenzierkommandos fortgesetzt. Um festzustellen, ob die Referenzierung abgeschlossen ist bzw. um den Status der Referenzierung auszulesen, existieren die Befehle **WaitRefFinished** und **IsAxisReferenced**.



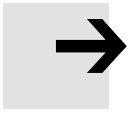
Syntax

RefAxisAsync(‹axis›:AXIS, OPT ‹refData›:REFDATA, OPT ‹timeout›:REAL)

Parameter	Bedeutung	Einheit
Axis	Ausgewählte Achse, die referenziert wird.	Enumeration A1, A2, bis A9
refData	Datensatz der Referenzfahrt, z. B. wie referenziert wird.	keine
timeout	Zeit für den Ablauf der Operation (danach kommt eine Fehlermeldung, die Referenzierung wird abgebrochen).	Sekunden

Tabelle 13.4 Parameter Befehl RefAxisAsync

Bei der Durchführung der Referenzfahrt ist darauf zu achten, dass die beteiligten Achsen freie Fahrt in ihren Bewegungen haben. Die Bewegung wird als Achsbewegung ausgeführt. Je nach Kinematik Typ kann es aufgrund der Achsanordnungen am Werkzeug zu unbekannten Bewegungen kommen.



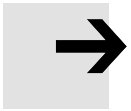
Hinweis

Die Parameter refData und timeout sind optional. Werden diese nicht angegeben, wird die aktuelle Ist-Position als Referenzposition übernommen (DS 402 - Methode 35).

Vorsicht

Bei der Referenzfahrt ist auf freie Fahrt aller Achsen zu achten. Ferner sind bei diesen Bewegungen geeignete Dynamikwerte zu wählen, um die Referenzfahrt sinnvoll auszuführen. Hohe dynamische Werte sind hier nicht sinnvoll.

Die Parameter und die sonstige Funktionsweise entspricht dem Befehl **RefAxis**.

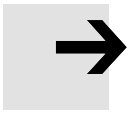


Hinweis

Wird die Referenzfahrtmethode 99 angegeben, werden die Parameter der Referenzfahrt, wie sie mit dem Festo Configuration Tool (FCT) im Antrieb abgelegt wurden, verwendet. Die Referenzfahrt wird dann so ausgeführt, wie bei Inbetriebnahme der einzelnen Achse. Alle andere Parameter wie offset, velSwitch, zeroSwitch und acc haben in diesem Fall keine Bedeutung.

13.3 Warten auf Ende der Referenzfahrt ‹WaitRefFinished›

Warten bis alle asynchron gestarteten Referenzfahrten abgeschlossen sind.



Syntax

WaitRefFinished() : BOOL

Der Befehl wartet, bis die asynchronen Referenzfahrten beendet wurden (es wird auf den Hauptlauf gewartet) oder bis ein Fehler bei einer Referenzfahrt aufgetreten ist.

Ist bei der Referenzfahrt kein Fehler aufgetreten, wird TRUE zurückgegeben, ansonsten FALSE.

Programm:

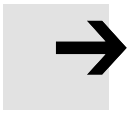
```
:
RefAxisAsync(A1, refdata0)
RefAxisAsync(A2, refdata0)
RefAxisAsync(A3, refdata0)
RefAxisAsync(A4)
boolReference := WaitRefFinished()
IF NOT boolReference THEN
   SetError("Error homing")
END_IF
```

Vorsicht

Bei der Ausführung des Makros RefAxisAsync ist die Verwendung des Makros WaitRefFinished zwingend nötig um die weitere Programmbearbeitung sicherzustellen. Wird nicht auf das Ende der Referenzfahrt gewartet, so können etwaige Befehle, bedingt durch die Vorausberechnung des Programms, Fehler erzeugen.

13.4 Status einer Achse abfragen ‹IsAxisReferenced›

Abfrage, ob eine Achse referenziert ist.



Syntax

IsAxisReferenced(axis : AXIS) : BOOL

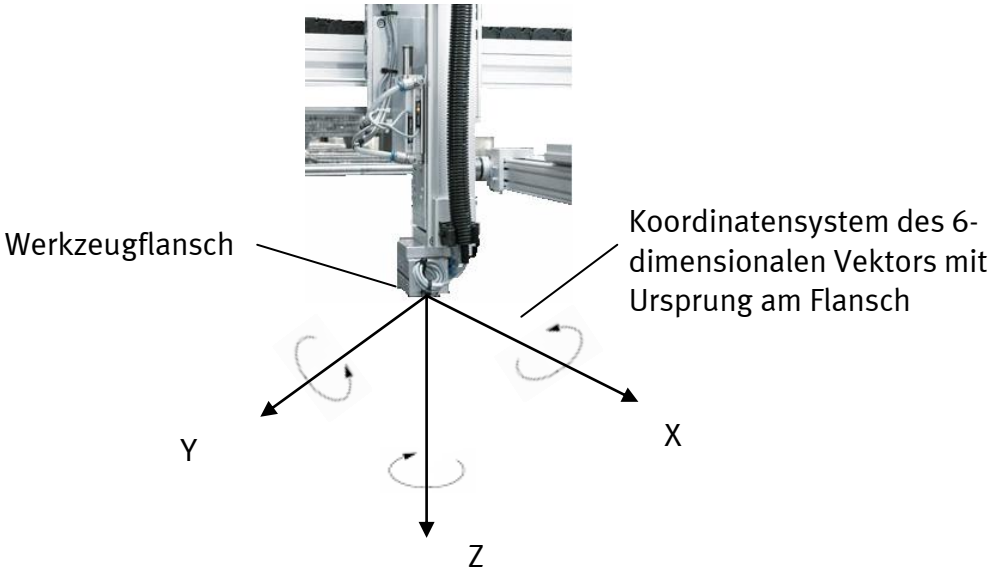
Parameter	Bedeutung	Einheit
Axis	Ausgewählte Achse, die abgefragt werden soll.	Enumeration A1, A2, bis A9

Tabelle 13.5 Parameter Befehl IsAxisReferenced

Ist die angegebene Achse referenziert, wird TRUE zurückgegeben, ansonsten FALSE.

14. Werkzeuge

Die CMXR Mehrachssteuerung bietet die Möglichkeit, die Längendaten eines Werkzeuges zu definieren. Diese Daten werden in Form eines 6-dimensionalen Vektors beschrieben. Somit kann dem Werkzeug neben den Abmaßen auch eine Orientierung mitgegeben werden. Diese Orientierung erfolgt nach der Euler ZYZ Konvention. Der Ursprung des Vektors ist dabei der Nullpunkt im Werkzeugflansch, das Ende bestimmt den Werkzeugendpunkt, der TCP (**T**ool **C**enter **P**oint) genannt wird. Mit den Werkzeugdaten wird das Werkzeugkoordinatensystem bestimmt.



14.1 Werkzeugdaten

14.1.1 Daten TCP-Vektor

Die Werkzeugdaten sind im strukturierten Datentyp TCPTOOL abgelegt. Dieser enthält die Daten der 6 Freiheitsgrade.

Aufbau:

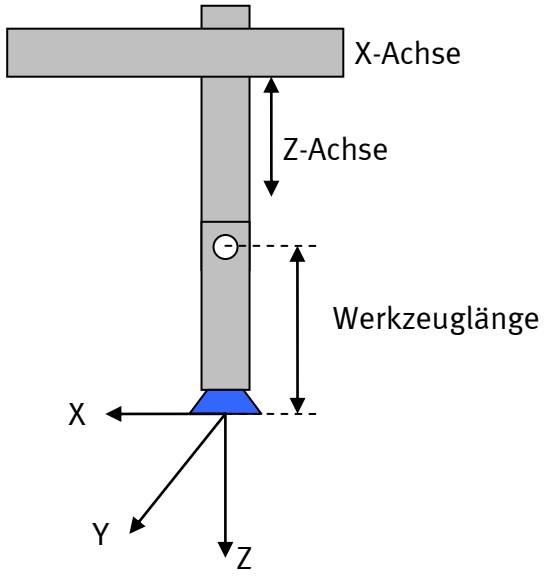
Datentyp TCPTOOL

x	: REAL	Verschiebung entlang der X-Achse
y	: REAL	Verschiebung entlang der Y-Achse
z	: REAL	Verschiebung entlang der Z-Achse
a	: REAL	Orientierungsangabe, Drehung um die Z-Achse
b	: REAL	Orientierungsangabe, Drehung um die verdrehte Y-Achse
c	: REAL	Orientierungsangabe, Drehung um die verdrehte Z-Achse

Die 3 translatorischen Werte X, Y und Z definieren den TCP im Raum, wenn alle Achsen in Ausgangslage stehen. Auf diese Weise wird das Werkzeugkoordinatensystem in den TCP verschoben. Dieses kann zusätzlich durch eine Orientierungsangabe (A-, B- und C-Parameter) in seiner Orientierung im Raum verdreht werden.

Beispiel TCP-Daten:

Ein kartesisches Portal besitzt am Ende der Z-Achse eine pneumatische Schwenkachse mit einem Vakuumgreifer. Das Werkzeug ist in Richtung der Z-Achse montiert. Die Orientierung des TCP ist unverändert zum ursprünglichen Werkzeugkoordinatensystem.



Es ergeben sich folgende TCP-Daten:

X = 0

Y = 0

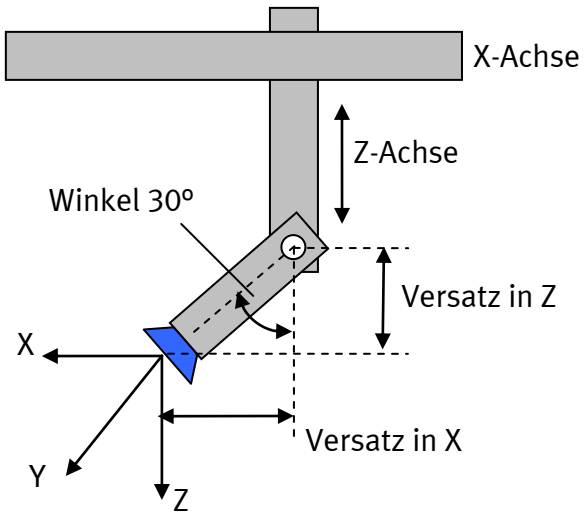
Z = Werkzeuglänge

A = 0

B = 0

C = 0

Nun ist das Werkzeug, das über eine pneumatische Schwenkachse verfügt, im Raum um 30 Grad geneigt. Der TCP berechnet sich über den Winkel der Schwenkbewegung.



Dabei ergeben sich folgende Daten:

X = Werkzeuglänge x sin(30°)

Y = 0

Z = Werkzeuglänge x cos(30°)

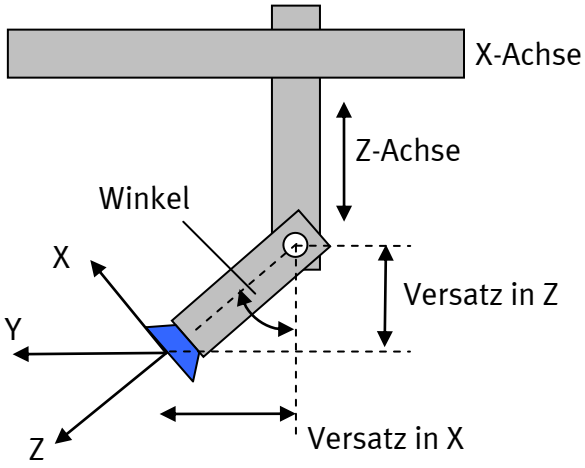
A = 0

B = 0

C = 0

Die Orientierung des Werkzeugkoordinatensystems ist unverändert. Wenn erforderlich, dann muss dieses über die Parameter A, B und C eingestellt werden.

Zusätzlich soll die Orientierung des Werkzeugkoordinatensystems in Richtung des gedrehten Werkzeuges zeigen. Zur Drehung wird die Euler-Konvention ZYZ angewandt.



Dabei ergeben sich folgende Daten:

X = Werkzeuglänge x sin(30°)

Y = 0

Z = Werkzeuglänge x cos(30°)

A = 0

B = 30

C = 0

Diese Werkzeugbeschreibungen werden in Variablen abgespeichert. Dabei können für ein Werkzeug beliebig viele TCP-Variable definiert werden, wobei nur ein Datensatz aktiv sein kann. Die verschiedenen Beschreibungen werden dann verwendet, wenn ein Werkzeug verschiedene Bezugspunkte besitzt und diese je nach Aufgabe im Betrieb gewechselt werden müssen.

Da die Werkzeugdaten im Speicher als Variable abgelegt sind, ist die Anzahl durch die Speichergröße bestimmt.

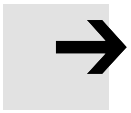
Für die richtige Zuordnung der TCP-Daten zum Werkzeug hat der Programmierer zu sorgen. Die CMXR Mehrachssteuerung kennt keinen Bezug der Werkzeugdaten zum physikalischen Werkzeug.

Vorsicht

Bei unpassenden oder falschen TCP-Daten besteht Kollisionsgefahr.

14.2 Aktivieren von Werkzeugdaten ‹Tool›

Mit folgendem Befehl können innerhalb eines FTL-Programms die Daten eines Tool Center Point (TCP) gesetzt werden. Dieser Tool-Befehl setzt neue TCP-Daten für die Kinematik. Somit wird der Arbeitspunkt der Kinematik verändert.



Syntax

Tool (‹ToolData› : TCPTOOL)

Die Daten für den TCP sind in der zu übergebenden Variable enthalten. Diese Daten werden im Satzvorlauf des FTL-Interpreters eingelesen und gehen ab dort in die weitere Bahnplanung der Bewegung ein. Bei allen folgenden Befehlen werden nun diese TCP-Daten berücksichtigt.

Parameter	Bedeutung	Einheit
ToolData	Werkzeugdaten	TCPTOOL

Tabelle 14.1 Parameter Befehl Tool

Der Aufruf des Befehls **Tool** bewirkt keine Verfahrbewegung, sondern nur ein Bekanntmachen der nun aktiven TCP-Daten. Im nächsten kartesischen Fahrbefehl werden diese Daten eingerechnet und in der Bewegungsausführung berücksichtigt.

Wird im FTL-Programm ein Tool-Befehl übersprungen, bzw. am Handbediengerät der Satzzeiger zur Programmausführung so positioniert, dass der zugehörige Tool Befehl nicht ausgeführt werden kann, können Gefahren für Mensch und Maschine entstehen. Die folgende Werkzeugorientierung könnte unpassend zur Bewegung sein, wobei Kollisionsgefahr besteht.

Warnung

Beim Ändern des Tools ergibt sich ein Sprung in der kartesischen Bahn des TCP. Wird am Handbediengerät der Programmzeiger so gesetzt, dass ein Tool Befehl umgangen wird, so kann dies bei kartesischen Bewegungen zu ungewollten Reaktionen führen!

14.2.1 Auswirkung von TCP-Daten

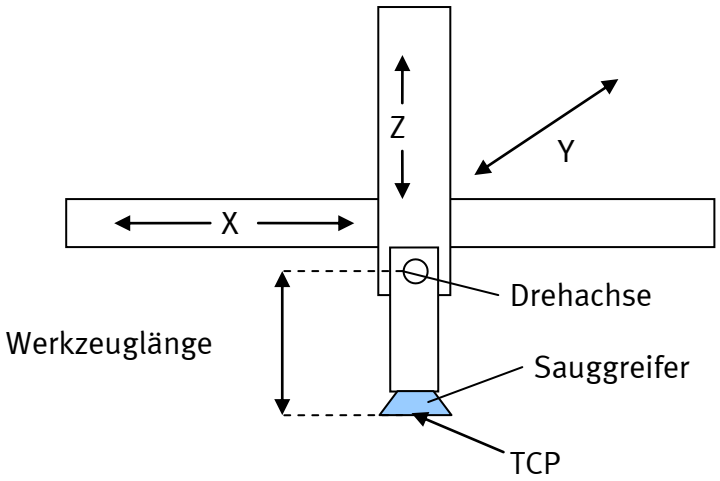
Die TCP-Daten werden im FTL-Programm über einen Befehl aktiviert und im Satzvorlauf des Interpreters eingelesen. Diese nun aktuellen TCP-Daten werden in die Bahnplanung der folgenden Bewegungsbefehle eingerechnet. Folgendes ist ein Beispiel, in dem das Verhalten von Werkzeugdaten mit der Anwendung einer pneumatischen Schwenkachse beschrieben wird.

Beispiel:

Ein Handhabungssystem besitzt am Werkzeugflansch eine pneumatische Schwenkachse. Mit Hilfe dieser Achse kann das Werkzeug auf eine feste Position geschwenkt werden. Durch diese Schwenkbewegung wird die Orientierung des Werkzeuges verändert. Damit nun die CMXR Mehrachssteuerung unter Berücksichtigung der Lage des TCP eine kartesische Bewegung im Raum berechnen kann, muss nach der Schwenkbewegung die neue Orientierung mitgeteilt werden.

Bedingt durch die zwei möglichen Orientierungen des Werkzeuges, gibt es 2 TCPs und somit 2 Datensätze tool1 und tool2.

Orientierung 1, Werkzeug senkrecht:

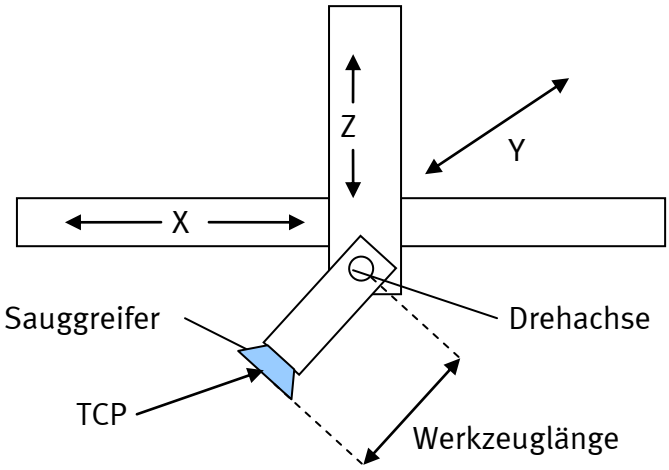


Die Werkzeuglänge des senkrechten Werkzeuges läuft entlang der Z- Achse des Werkzeugkoordinatensystems. Somit besitzt der TCP nur eine Translation in Richtung der Z-Achse, die Orientierungsangaben sind 0.

Werkzeugdaten für tool1:

X	= 0	Verschiebung entlang der X-Achse
Y	= 0	Verschiebung entlang der Y-Achse
Z	= Werkzeuglänge	Verschiebung entlang der Z-Achse
A	= 0	Orientierung nach Euler, Drehung um die Z-Achse
B	= 0	Orientierung nach Euler, Drehung um die verdrehte Y-Achse
C	= 0	Orientierung nach Euler, Drehung um die verdrehte Z-Achse

Orientierung 2, Werkzeug geschwenkt:



Durch die Schwenkbewegung wandert der TCP zur Seite, das Werkzeug hat nun eine veränderte Orientierung.

Bei der abgebildeten Kinematik handelt es sich um ein kartesisches System. Bei diesem ist die Lage des kartesischen Koordinatensystems der Grundachsen X, Y und Z deckungsgleich mit dem kartesischen Werkzeugkoordinatensystem. Wendet man nun zur Bestimmung der Orientierung die Rechte-Hand-Regel an, so ergibt sich eine Drehung des Werkzeuges um die Y-Achse in positiver Richtung.

Werkzeugdaten für tool2:

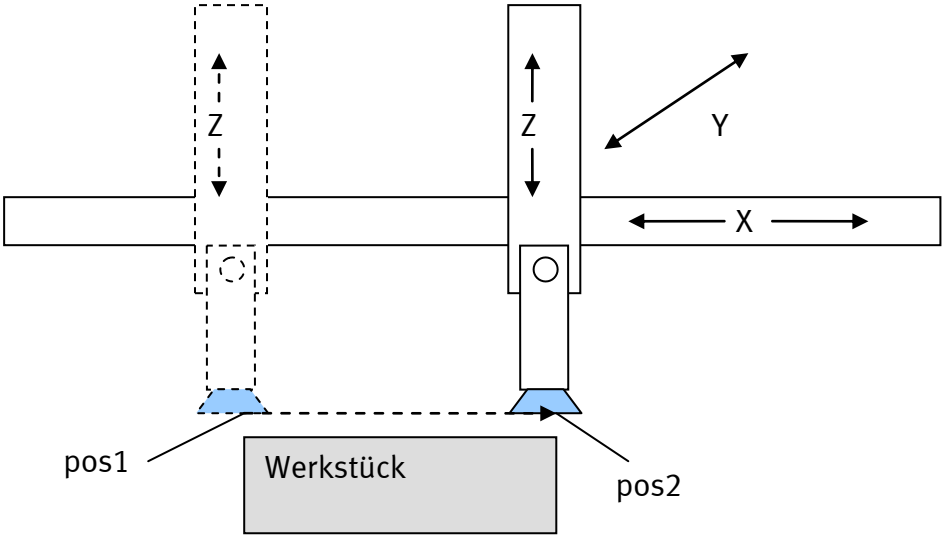
X	= Länge x sin(30°)	Verschiebung entlang der X-Achse
Y	= 0	Verschiebung entlang der Y-Achse
Z	= Länge x cos(30°)	Verschiebung entlang der Z-Achse
A	= 0	Orientierung nach Euler, Drehung um die Z-Achse
B	= 30	Orientierung nach Euler, Drehung um die verdrehte Y-Achse
C	= 0	Orientierung nach Euler, Drehung um die verdrehte Z-Achse

Programmbeispiel:

Unsere kartesische Kinematik soll nun mit senkrechtem Werkzeug von einer Position 1 auf eine Position 2 fahren. Danach wird das Werkzeug durch die Schwenkachse geschwenkt. Das geschwenkte Werkzeug muss nun die Position 2 anfahren.

Anfahren von pos1 nach pos2:

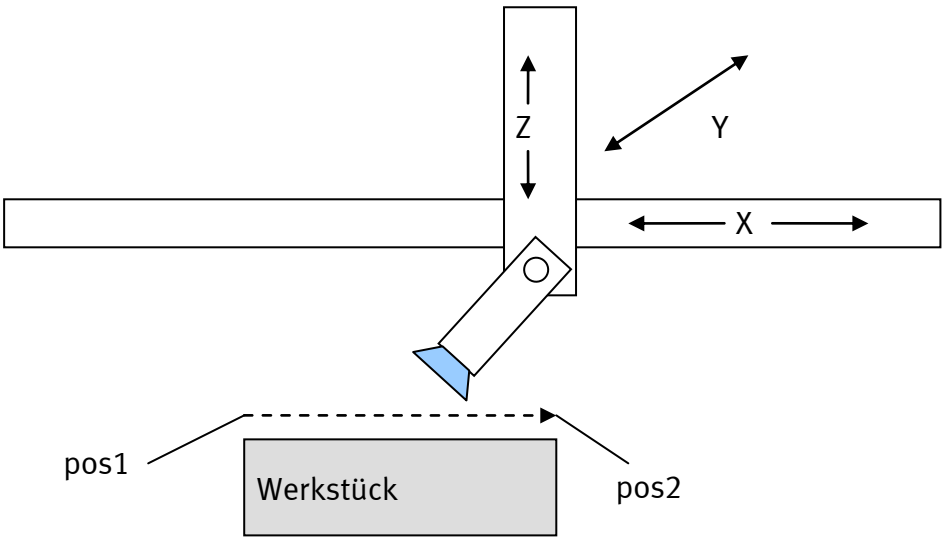
```
Tool(tool1)
Lin(pos1)
Lin(pos2)
```



Bei den kartesischen Fahrbefehlen von pos1 auf pos2 berücksichtigt die Steuerung automatisch die aktiven Werkzeugdaten tool1.

Schwenken der Schwenkachse:

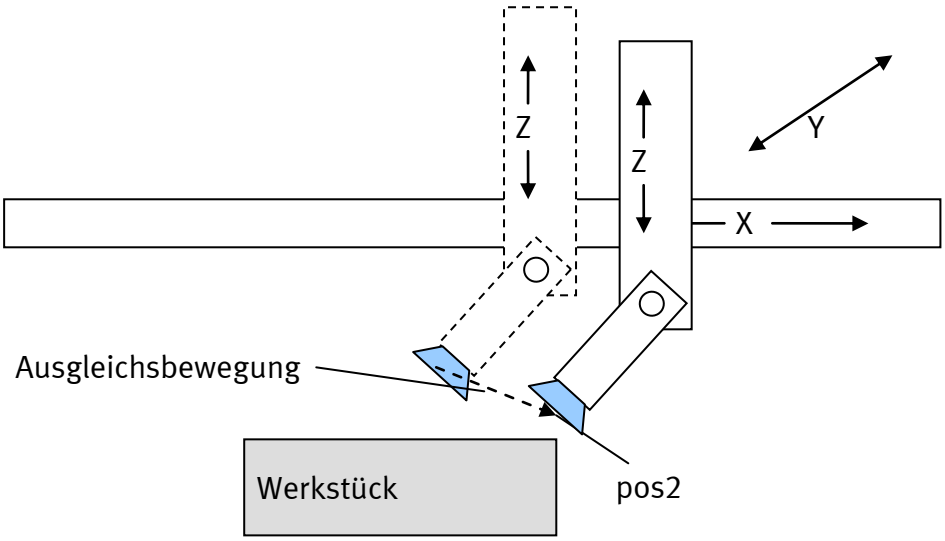
```
:
Tool(tool1)
Lin(pos1)
Lin(pos2)
CALL Rotate()          // Aufruf Unterprogramm
:
```



Die Schwenkbewegung hat zur Folge, dass sich nun der Tool Center Point (TCP) nach oben bewegt. Aufgrund der pneumatischen Achse, kann die Steuerung keine automatischen Ausgleichsbewegungen durchführen. Um nun die Position 2 anzufahren, muss die neue Orientierung des Werkzeuges definiert werden.

Erneutes Anfahren von pos2 mit tool2:

```
:
Tool(tool1)
Lin(pos1)
Lin(pos2)
CALL Rotate()          // Aufruf Unterprogramm
Tool(tool2)
Lin(pos2)
:
```



Das erneute Anfahren von Position 2 hat die Folge, dass der TCP mit seinen neuen Daten auf die Position 2 ausgerichtet wird. Je nach Fall, kann dabei eine Ausgleichsbewegung mit max. allen Achsen stattfinden.

Hinweis

Bei Verwendung einer elektrischen Schwenkachse die als Freiheitsgrad in die kartesische Kinematik eingearbeitert ist, wir im Gegensatz zur pneumatischen Schwenkachse keine Änderung der Werkzeugorientierung erforderlich.

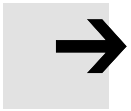
Bei der elektrischen Schwenkachse wird die gewünschte Orientierung innerhalb eines fahrbefehls programmiert.

Die CMXR Steuerung berechnet dann automatisch die Position der Achsen unter Berücksichtigung der programmierten Orientierung.

15. PROFIBUS-Schnittstelle

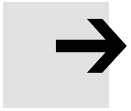
Über die PROFIBUS-Schnittstelle kann die CMXR von einer übergeordneten Steuerung (SPS/IPC) gesteuert werden. Weiterhin können Daten in Form von Variablen geschrieben und gelesen werden. All diese Daten sind systemglobale Daten und stehen allen FTL- Programmen zur Verfügung. Die Kommunikation zur übergeordneten Steuerung verläuft zyklisch im Updatezyklus des PROFIBUS.

Beim Neustart des Systems sind alle Werte auf Null gesetzt. Etwaige Daten, die für die Abarbeitung der Programme benötigt werden, müssen von der übergeordneten Steuerung vor dem Programmstart gesendet werden.



Hinweis

Werden in einem Programm konsistente Daten benötigt, ist vor deren Verarbeitung dafür zu sorgen, dass diese auf z. B. lokale Daten kopiert werden.



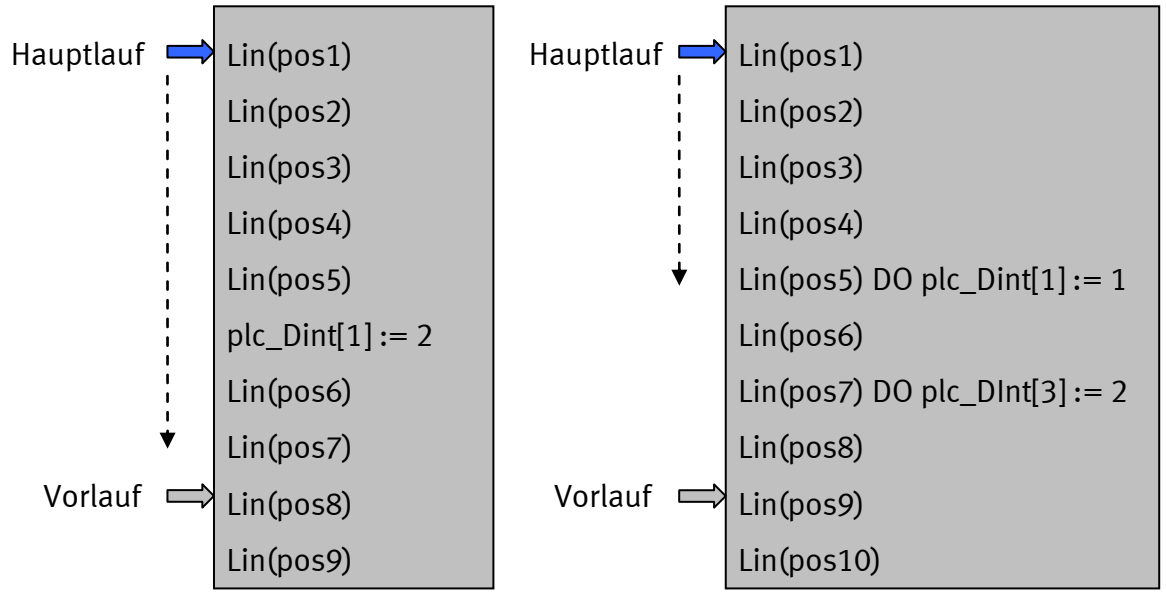
Hinweis

Die Schnittstellendaten sind nicht gepuffert und haben nach einem Neustart des Systems alle den Wert Null. Etwaige Daten, die zur Abarbeitung benötigt werden, sind vor dem Start zu schreiben.

15.1 Hinweise zur Signalverarbeitung

Die in den folgenden Kapiteln behandelten FTL Variablen der SPS Schnittstelle werden wie andere Variable immer in der Vorausberechnung (Satzvorlauf) des FTL Programms berechnet. Müssen Variablen zur Laufzeit der aktiven Programmzeile (Hauptlauf) ausgeführt werden, so sind zusätzliche Maßnahmen erforderlich. Eine Möglichkeit ist die Verwendung der DO Anweisung.

Die folgende Grafik erläutert die Unterschiede der Signalverarbeitung:



Variable wird im Satzvorlauf bearbeitet.

Verwendung der DO Anweisung, Variable wird im Hauptlauf bearbeitet.

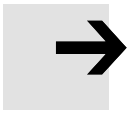
15.2 Boolesche Ein- und Ausgänge

Die Schnittstelle umfasst 16 digitale Ein- und Ausgangssignale, die im Weiteren immer aus Sicht des CMXR gesehen werden. Sie sind in Form einer booleschen Variable auf der Steuerung verfügbar. Diese booleschen Signale werden zyklisch mit der übergeordneten Steuerung ausgetauscht.

Hinweis

Die booleschen Variablen werden über den Zyklus des PROFIBUS automatisch zur übergeordneten Steuerung übertragen bzw. von dieser gelesen.

Die booleschen Variablen sind in einem Array abgelegt, auf das über den Index 0 bis 15 zugegriffen werden kann.



Syntax

plc_InBool [‹Arrayindex›] : BOOL

plc_OutBool [‹Arrayindex›] : BOOL

In der Arrayvariablen **plc_InBool** sind alle Eingangssignale gespeichert. Die Arrayvariable **plc_OutBool** enthält alle Ausgangsdaten.

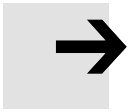
Beispiel:

```
:
plc_OutBool[9] := FALSE          // Bit 9 zur PLC auf FALSE
IF plc_InBool[5] THEN            // prüfen Bit 5 von PLC
  Lin(pos1)
  Lin(pos2)
END_IF
plc_OutBool[9] := TRUE           // Bit 9 zur PLC auf TRUE
:
```

Beispiele mit Satzvorlauf, mit DO Anweisung

15.3 Integer-Variablen 32 Bit

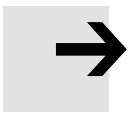
Die Schnittstelle bietet 256 Integervariablen vom Datentyp DINT, der 32 Bit umfasst. Diese Variablen unterliegen keinem zyklischen Datenaustausch und können von der externen Steuerung nach Bedarf geschrieben und gelesen werden.



Hinweis

Die Integervariablen werden nicht automatisch von der übergeordneten Steuerung gelesen bzw. an diese übermittelt. Wenn erforderlich, dann müssen diese von der übergeordneten Steuerung gesendet bzw. gelesen werden.

Die Integervariablen sind in einem Array abgelegt, auf das über den Index 0 bis 255 zugegriffen werden kann.



Syntax

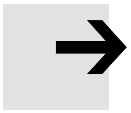
plc_Dint [‹Arrayindex›] : DINT

Beispiel:

```
:
IF plc_Dint[3] = 13 THEN
:
:
END_IF
```

15.4 Positionen

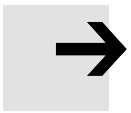
Die Schnittstelle unterstützt zwei Positionsdatentypen. Es können Achspositionen und kartesische Positionen von der externen Steuerung zur CMXR Mehrachssteuerung gesendet werden. Die maximale Anzahl der Positionen umfasst dabei 256 Achs- und 256 kartesische Positionen. Diese Variablen unterliegen keinem zyklischen Datenaustausch und können von der externen Steuerung nach Bedarf geschrieben und gelesen werden.



Hinweis

Die Positionsvariablen werden nicht automatisch von der übergeordneten Steuerung gelesen bzw. an diese übermittelt. Wenn erforderlich, dann müssen diese von der externen Steuerung gesendet bzw. gelesen werden.

Die Positionsvariablen sind in Arrays abgelegt. Über den Index 0 bis 255 kann auf diese zugegriffen werden.



Hinweis

plc_AxisPos [‹Arrayindex›] : AXISPOS

plc_CartPos [‹Arrayindex›] : CARTPOS

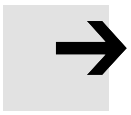
Die Schnittstellenvariable **plc_AxisPos** enthält 256 Positionen vom Datentyp AXISPOS, die Variable **plc_CartPos** enthält 256 Positionen von Datentyp CARTPOS.

Beispiel:

```
:
Ptp(plc_AxisPos[17])
Lin(plc_AxisPos[18])
Lin(plc_AxisPos[19])
Lin(plc_CartPos[1])
:
```

15.5 Referenzsysteme

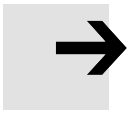
Maximal 16 Referenzsysteme können über die externe Schnittstelle vorgegeben werden. Diese Variablen sind vom Datentyp REFSYSDATA und können mit dem Befehl **SetRefSys** verwendet werden. Diese Variablen unterliegen keinem zyklischen Datenaustausch und können von der externen Steuerung nach Bedarf geschrieben und gelesen werden.



Hinweis

Die Referenzsysteme werden nicht automatisch von der übergeordneten Steuerung gelesen bzw. an diese übermittelt. Wenn erforderlich, dann müssen diese von der externen Steuerung gesendet bzw. gelesen werden.

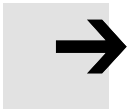
Die Referenzsysteme sind in einem Array abgelegt, auf das über den Index 0 bis 15 zugegriffen werden kann.



Syntax

plc_RefSys [‹Arrayindex›] : REFSYSDATA

Der Datentyp REFSYSDATA bietet die Möglichkeit, ein Referenzsystem additiv auf ein anderes zu beziehen. Die Schnittstelle unterstützt dies ebenfalls, jedoch nur innerhalb der Schnittstelle selbst. D. h. es kann kein Bezug auf im System bestehende Referenzsysteme hergestellt werden. Der Bezug auf ein anderes Referenzsystem basiert auf einer Nummernvorgabe 0 bis 15 durch die externe Steuerung. Bei einer Vorgabe von -1 wird der Bezug zum Weltkoordinatensystem der Kinematik hergestellt.



Hinweis

Referenzsysteme, die über die externe Schnittstelle vorgegeben werden, können nur innerhalb der Referenzsysteme der Schnittstelle verlinkt werden. Ein Bezug auf ein Referenzsystem, das außerhalb der Schnittstelle definiert wurde, ist nicht möglich.

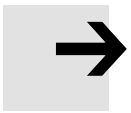
Beispiel:

```
:
Lin(pos2)
SetRefSys(plc_RefSys[3])
Lin(pos3)
Lin(pos4)
:
```

15.6 Programmierter Halt ‹ProgHold›

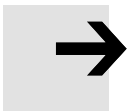
Dieser Befehl arbeitet in Verbindung mit dem Signal HALTENA der SPS Schnittstelle. Er eignet sich für Test- oder Inbetriebnahmezwecke und kann an beliebiger Programmzeile eingefügt werden, um dort auf Anforderung das Programm anzuhalten.

Ist das Signal HALTENA auf der Schnittstelle aktiviert, d. h. es hat den Zustand TRUE, wird bei einem Aufruf des Makros ProgHold das Bewegungsprogramm angehalten. Dabei wird nur das Programm angehalten, in dem sich das Makro ProgHold befindet. Andere Programme z. B. parallele Programme, werden weiter abgearbeitet. Setzt die SPS das Signal HALTENA auf FALSE, so wird das angehaltene Programm fortgesetzt.



Hinweis

Werden mit dem Makro ProgHold mehrere Programme angehalten (z. B. parallele Programme), so werden diese mit dem Signalzustand FALSE gemeinsam wieder anlaufen. Ein selektives Anlaufen der einzelnen angehaltenen Programme ist nicht möglich.



Syntax

ProgHold ()

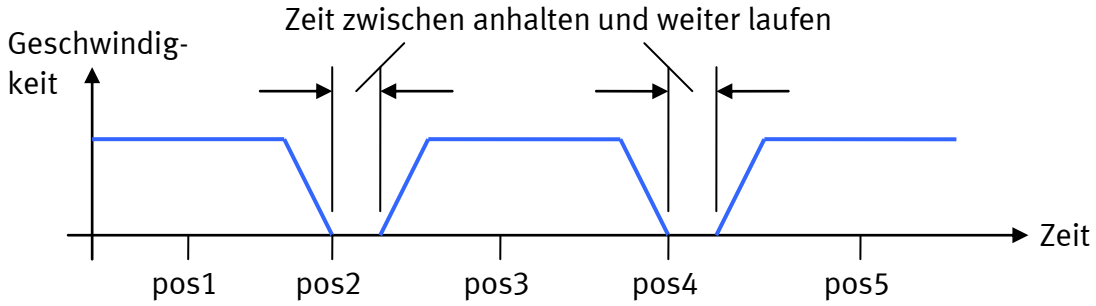
Zum Aufruf des Makros sind keine Parameter notwendig.

Beispiel:

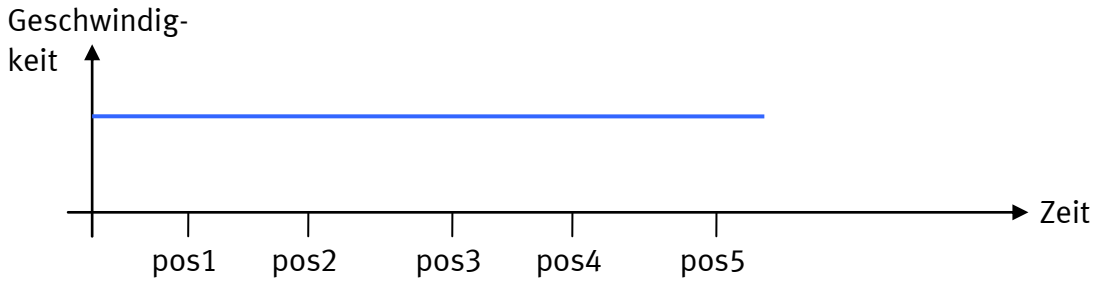
```
:
OvlVel(100)     // volles Überschleifen
Lin(pos1)
Lin(pos2)
ProgHold()      // programmierter Halt
Lin(pos3)
Lin(pos4)
ProgHold()      // programmierter Halt
Lin(pos5)
:
```

Dieses Beispielprogramm fährt verschiedene Positionen an, wobei ein Geschwindigkeitsüberschleifen mit 100 % eingestellt ist. Wenn das Signal zum programmierten Halt gesetzt ist, wird an diesen Stellen angehalten. Dabei ergibt sich folgender Geschwindigkeitsverlauf:

Verlauf mit programmiertem Halt:



Verlauf ohne programmierten Halt:



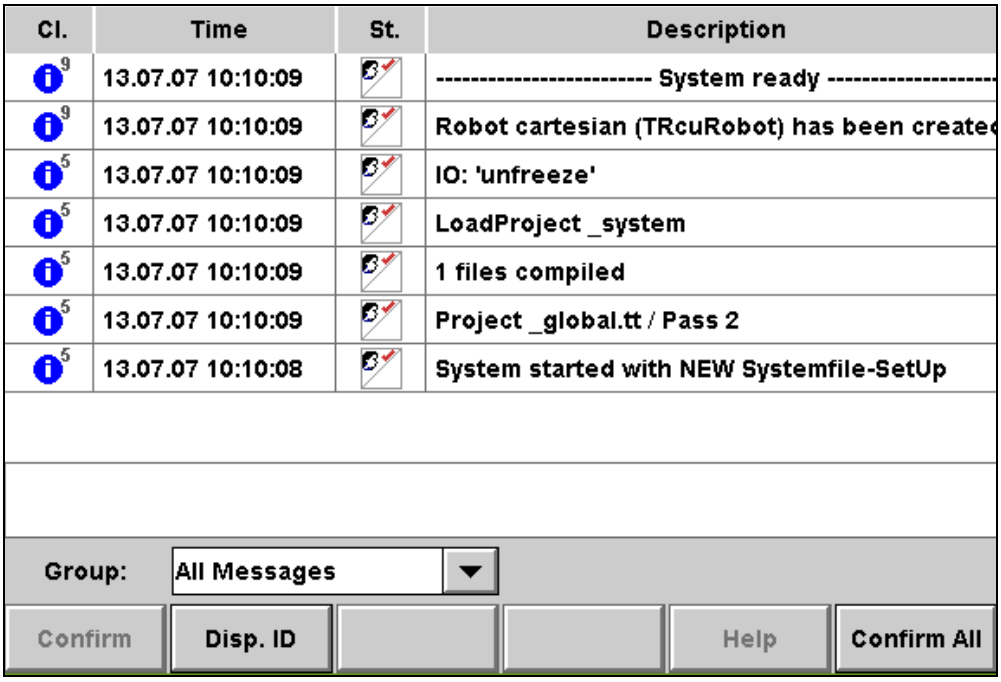
16. Meldesystem

Aus dem Bewegungsprogramm können Meldungen generiert werden. Es gibt folgende Meldungsarten:

- Information
- Warnung
- Fehler

Diese Meldungen werden in den Meldespeicher der Steuerung eingetragen und entsprechend archiviert. Zurückgesetzt (gelöscht) werden die Meldungen durch Quittieren am Handbediengerät oder Quittieren über eine externe Steuerung.

Abbildung des Meldespeichers am Handbediengerät:



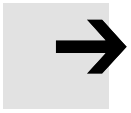
16.1 Meldetexte

Die Programmierung in FTL (Festo Teach Language) ermöglicht das Absetzen von Informations-, Warn- und Fehlermeldungen aus dem Bewegungsprogramm. Die Meldungstexte werden vom Anwender selbst als Zeichenkette (String) frei definiert. Diese Zeichenkette kann auch variable Einträge in Form von maximal 2 Parametern (z. B. DINT, REAl, String, BOOL) enthalten. Die 2 optionalen Parameter werden über eine Angabe eines Prozentzeichens und einer Nummer im Meldetext platziert.

%1 bedeutet 1. optionaler Parameter

%2 bedeutet 2. optionaler Parameter

Werden Typen wie Achspositionen angegeben, so wird lediglich der Variablenname in den String eingefügt. Strukturierte Datentypen können nicht in einem Meldetext dargestellt werden.



Hinweis

Bei der Übergabe der Variablentypen DINT, REAL und String wird der Inhalt der Variablen in den Meldetext eingefügt. Beim Variablentyp BOOL wird je nach Zustand der Text TRUE oder FALSE in den Meldetext eingefügt.

Beispiel:

In einem Programm werden zwei Variable angelegt und mit Werten vorbelegt. Diese Werte werden mit dem Makro **SetInfo** ausgegeben.

Variablen:

```
param1 : DINT := 7
param2 : REAL := 3.48
```

Programmcode:

```
SetInfo("Sensor %1, pressure %2 bar", param1, param2)
```

Es wird folgender Infotext angezeigt:“Sensor 7, pressure 3,48 bar“.

16.2 Information ‹SetInfo›

Der Befehl **SetInfo** setzt eine Informationsmeldung in das Meldesystem ab.

Syntax

SetInfo(‹text› : STRING,
OPT ‹param1› : ANY,
OPT ‹param2› : ANY)

Parameter	Bedeutung	Einheit
text	Text der Infomeldung	STRING
param1	1. möglicher Parameter	ANY
Param2	2. möglicher Parameter	ANY

Tabelle 16.1 Parameter Befehl SetInfo

Die Zusammensetzung des Meldetextes ist in Kapitel 16.1 Meldetext auf Seite 137 beschrieben.

Eine Informationsmeldung wird mit dem Symbol im Fehlerspeicher der CMXR Mehrachssteuerung gekennzeichnet.

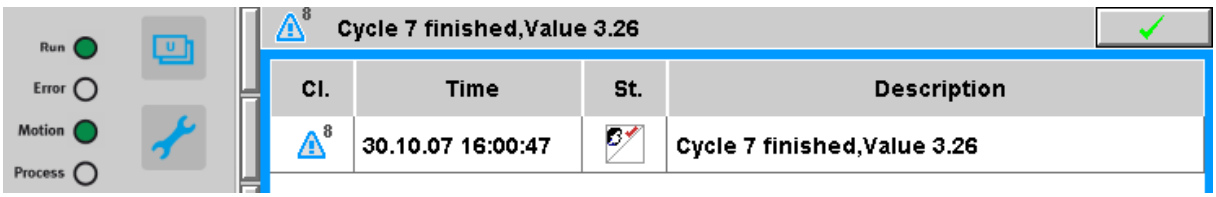
Hinweis

Eine Informationsmeldung hat keine Auswirkung auf die Bewegung. Sie dient lediglich der Information.

Beispiel:

```
pressure := Sensor.Read();    // lesen eines Druckwertes
cycle    := cycle + 1         // Zyklus zählen
SetInfo ("Cycle %1 finished, Value %2", cycle, pressure)
```

Anzeige am Handbediengerät:



16.3 Warnung ‹SetWarning›

Der Befehl **SetWarning** setzt eine Warnmeldung in das Meldesystem ab.

Syntax

SetWarning(‹text› : STRING,
OPT ‹param1› : ANY,
OPT ‹param2› : ANY)

Parameter	Bedeutung	Einheit
text	Text der Warnmeldung	STRING
param1	1. möglicher Parameter	ANY
Param2	2. möglicher Parameter	ANY

Tabelle 16.2 Parameter Befehl SetWarning

Die Zusammensetzung des Meldetextes ist in Kapitel 16.1 Meldetext auf Seite 137 beschrieben.

Eine Warnmeldung wird mit dem Symbol ⚠ im Fehlerspeicher der CMXR Mehrachssteuerung gekennzeichnet.

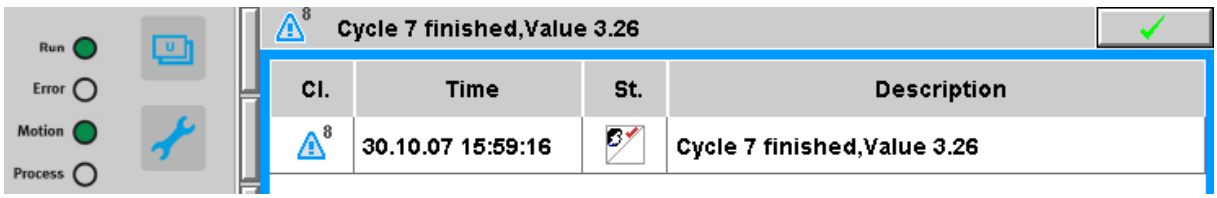
Hinweis

Eine Warnmeldung hat keine Auswirkung auf die Bewegung. Sie dient lediglich der Information.

Beispiel:

```
pressure := Sensor.Read();    // lesen eines Druckwertes
cycle    := cycle + 1         // Zyklus zählen
SetWarning("Cycle %1 finished, Value %2",cycle, pressure)
```

Anzeige am Handbediengerät:



16.4 Fehlermeldung ‹SetError›

Der Befehl **SetError** setzt eine Fehlermeldung in das Meldesystem ab. Dieser Befehl wirkt sich auf die Bearbeitung des Programms aus, es wird gestoppt. Etwaige Bewegungen werden angehalten, nach Quittierung der Fehlermeldung wird das Programm / die Bewegung fortgesetzt.

Syntax

SetError(‹text› : STRING,
OPT ‹param1› : ANY,
OPT ‹param2› : ANY)

Parameter	Bedeutung	Einheit
Text	Text der Fehlermeldung	STRING
param1	1. möglicher Parameter	ANY
Param2	2. möglicher Parameter	ANY

Tabelle 16.3 Parameter Befehl SetError

Eine Fehlermeldung wird mit dem Symbol im Fehlerspeicher der CMXR Mehrachssteuerung gekennzeichnet.

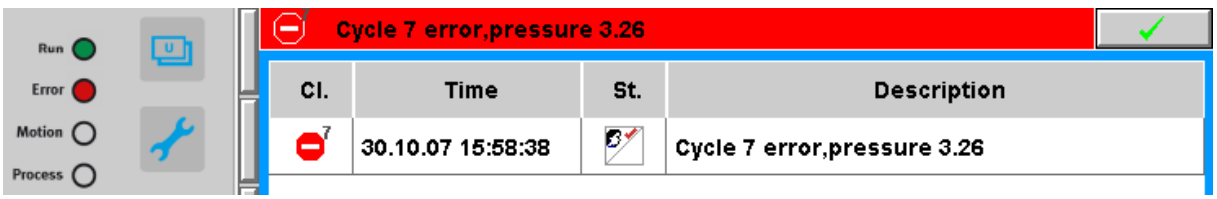
Hinweis

Die Ausgabe einer Fehlermeldung führt zu einer Unterbrechung der Bewegung. Die Kinematik kann erst nach Quittieren der Fehlermeldung weitergefahren werden.

Beispiel:

```
pressure := Sensor.Read();    // lesen eines Druckwertes
cycle    := cycle + 1         // Zyklus zählen
SetError("Cycle %1 error, pressure %2", cycle, pressure)
```

Anzeige am Handbediengerät:

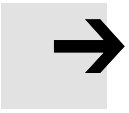


Bei einer Fehlermeldung wird der Fehlertext zusätzlich in der Kopfzeile am Handbediengerät angezeigt. Außerdem leuchtet die Error LED am Handbediengerät rot.

17. Funktionen

17.1 Aktuelle Position lesen ‹ReadActualPos›

Auslesen der aktuellen Position der Kinematik.



Syntax

ReadActualPos (‹Pos› : POSITION)

Die aktuelle Position des Roboters wird in die übergebene Variable geschrieben. Ist die übergebene Variable vom Typ CARTPOS, wird die Position als kartesischer Wert abgelegt. Wird eine Variable vom Typ AXISPOS übergeben, erfolgt die Positionsspeicherung in Achskoordinaten.

Parameter	Bedeutung	Einheit
Pos	Zielvariable der gelesenen Position	AXISPOS oder CARTPOS

Tabelle 17.1 Parameter Befehl ReadActualPos

Vorsicht

Ein Positionswert, der in einer Variablen gespeichert wurde, bleibt nur solange erhalten, wie das Programm bzw. das Projekt geladen ist. Es erfolgt kein Sichern der Variablenwerte in die Datendatei auf der Speicherkarte. Nach dem Abwählen des Programms/Projektes sind die Werte verloren.
Zum Sichern der Position kann das Makro SavePosition verwendet werden.

Beispiel:

Das Beispielprogramm liest die aktuelle Position ein, fährt die programmierten Bewegungsbefehle ab und kehrt am Ende zur eingelesenen Position zurück.

Variable:

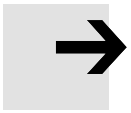
```
startPos : AXISPOS := (0, 0, 0, 0, 0, 0, 0, 0, 0)
axis0    : AXISPOS := (0, 0, 0, 0, 0, 0, 0, 0, 0)
axis1    : AXISPOS := (60, -120, 0, 0, 0, 0, 0, 0, 0)
axis2    : AXISPOS := (-120, -120, -250, 0, 0, 0, 0, 0, 0)
```

Programm:

```
ReadActualPos(startPos)
PTP(axis0)
PTP(axis1)
PTP(axis2)
PTP(startPos)
```

17.2 Zielposition lesen ‹ReadTargetPos›

Auslesen der programmierten Zielposition der Kinematik.



Syntax

ReadTargetPos (‹Pos› : POSITION)

Die aktuell programmierte Zielposition des Roboters wird in die übergebene Variable geschrieben. Ist die übergebene Variable vom Typ CARTPOS, wird die Position als kartesischer Wert abgelegt. Wird eine Variable vom Typ AXISPOS übergeben, erfolgt die Positionsspeicherung in Achskoordinaten.

Parameter	Bedeutung	Einheit
Pos	Zielvariable der gelesenen Position	AXISPOS oder CARTPOS

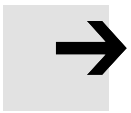
Tabelle 17.2 Parameter Befehl ReadTargetPos

Vorsicht

Ein Positionswert, der in einer Variablen gespeichert wurde, bleibt nur solange erhalten, wie das Programm bzw. das Projekt geladen ist. Es erfolgt kein Sichern der Werte in die Datendatei auf der Speicherkarte. Nach dem Abwählen des Programms, Projektes sind die Werte verloren.
Zum Sichern der Position kann das Makro SavePosition verwendet werden.

17.3 Positionswert dauerhaft speichern ‹SavePosition›

Das Makro speichert einen Positionswert dauerhaft auf der Speicherkarte.



Syntax

SavePosition (‹Pos› : POSITION)

Mit dem Makro **SavePosition** erfolgt ein Sichern der Werte aus der angegebenen Variable ‹Pos› in die Datendatei auf der Speicherkarte. Die Position kann somit bei einem Neustart der Steuerung wiederhergestellt werden.

Parameter	Bedeutung	Einheit
Pos	zu speichernde Position	AXISPOS oder CARTPOS

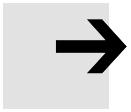
Tabelle 17.3 Parameter Befehl SavePosition

Vorsicht

Ein häufiges Zurückschreiben von Positionen auf die Speicherkarte verkürzt deren Lebensdauer. Das Makro SavePosition darf keinesfalls zyklisch aufgerufen werden. Es kann zum gelegentlichen Einrichten der Applikation verwendet werden.

17.4 Lesen der Systemzeit ‹Time›

Lesen der Systemzeit in Sekunden seit dem 01.01.1970 um 00:00 Uhr.



Syntax

(‹Zeitwert› : DINT) := **Time** ()

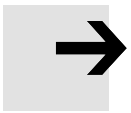
Dieser Befehl liest die aktuelle Systemzeit der Steuerung aus und gibt ihn als DINT-Wert zurück.

Beispiel:

```
value := Time()     // aktuelle Systemzeit lesen
```

17.5 Umwandeln Zeit in Text ‹TimeToStr›

Umwandeln einer Zeit in einen Text.



Syntax

(‹Zeitstring› : STRING) := **TimeToStr** (OPT sysTime : DINT)

Dieser Befehl kann eine Zeit (Parameter sysTime) in einen formatierten Text mit dem Format „DDD mon dd hh:mm:ss yyyy“ umwandeln. Ohne Parameter sysTime wird die aktuelle Systemzeit formatiert zurückgegeben.

Der Parameter sysTime gibt die Zeit in Sekunden seit dem 01.01.1970 um 00:00 Uhr an.

Parameter:

Parameter	Bedeutung	Einheit
sysTime	Optionale Angabe, Zeitwert, der gewandelt werden soll.	Sekunden

Tabelle 17.4 Parameter Funktion TimeStr

Beispiel:

```
str_Time := TimeToStr()    // aktuelle Systemzeit lesen
```

Rückgabe:

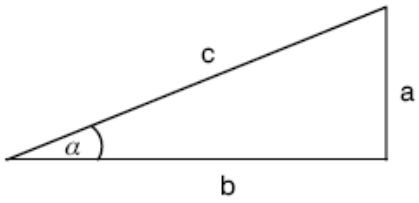
```
str_Time = „Mon Feb 13 11:23:44 2006“
```

17.6 Sinus ‹SIN›, ‹ASIN›

Die Sinus-Funktion stellt den mathematischen Zusammenhang zwischen einem Winkel und den Seiten in einem rechtwinkligen Dreieck her.

Für Sinus gilt:

sin(alpha) = Gegenkathete / Hypotenuse



sin(alpha) = a / c

Die Angabe des Winkels erfolgt in der Einheit Grad.

Syntax

(‹Sinuswert› : REAL) := **SIN**(‹Winkel› : REAL)

Verwendung:

Die Sinus-Funktion dient zur Berechnung von unbekannten Strecken in einem rechtwinkligen Dreieck. Ist der Winkel bekannt und entweder die Gegenkathete oder die Hypotenuse, so lässt sich die unbekannte Strecke berechnen.

Soll jedoch der Winkel berechnet werden, so stellt die CMXR die Funktion des Arcus Sinus bereit. Diese Funktion berechnet aus der Gegenkathete und der Hypotenuse den eingeschlossenen Winkel, z. B. Alpha.

Syntax

(‹Winkel in Grad› : REAL) := **ASIN**(‹Sinuswert›)

Die Angabe des Winkels erfolgt in Grad.

Beispiel:

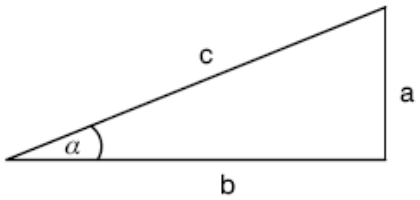
```
a     := 30              // Gegenkathete
alpha := 23.5            // Winkel Alpha
c     := a / SIN(alpha)  // Berechnung der Hypotenuse
a     : = 45.89          // Gegenkathete
c     := 145.67          // Hypotenuse
value := a / c
alpha := ASIN(value)     // Berechnung des Winkels
```

17.7 Cosinus ‹COS›, ‹ACOS›

Die Cosinus-Funktion stellt den mathematischen Zusammenhang zwischen einem Winkel und den Seiten in einem rechtwinkligen Dreieck her.

Für Cosinus gilt:

cos(alpha) = Ankathete / Hypotenuse



cos(alpha) = b / c

Die Angabe des Winkels erfolgt in der Einheit Grad.

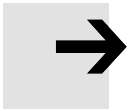
Syntax

(‹Cosinuswert› : REAL) := **COS**(‹Winkel›)

Verwendung:

Die Cosinus-Funktion dient zur Berechnung von unbekannten Strecken in einem rechtwinkligen Dreieck. Ist der Winkel bekannt und entweder die Ankathete oder die Hypotenuse, so lässt sich die unbekannte Strecke berechnen.

Soll jedoch der Winkel berechnet werden, so stellt die CMXR die Funktion des Arcus Cosinus bereit. Diese Funktion berechnet aus der Ankathete und der Hypotenuse den eingeschlossenen Winkel z. B. Alpha.



Syntax

(‹Winkel in Grad› : REAL) := **ACOS**(‹Cosinuswert›)

Die Rückgabe des Winkels erfolgt in Grad.

Beispiel:

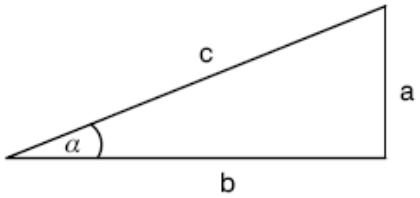
```
b     := 30              // Ankathete
alpha := 23.5            // Winkel Alpha
c     := b / COS(alpha)  // Berechnung der Hypotenuse
b     := 45.89           // Ankathete
c     := 145.67          // Hypotenuse
value := b / c
alpha := ACOS(value)     // Berechnung des Winkels
```

17.8 Tangens ‹TAN›, ‹ATAN›

Die Tangens-Funktion stellt den mathematischen Zusammenhang zwischen einem Winkel und den Seiten in einem rechtwinkligen Dreieck her.

Für Tangens gilt:

tan(alpha) = Gegenkathete / Ankathete



tan(alpha) = a / b

Die Angabe des Winkels erfolgt in der Einheit Grad.

Syntax

(‹Tangenswert› : REAL) := **TAN**(‹Winkel›)

Verwendung:

Die Tangens-Funktion dient zur Berechung von unbekannten Stecken in einem rechtwinkligen Dreieck. Ist der Winkel bekannt und auch entweder die Ankathete oder die Gegenkathete, so lässt sich die unbekannte Strecke berechnen.

Soll jedoch der Winkel berechnet werden, so stellt die CMXR die Funktion des Arcus Tangens bereit. Diese Funktion berechnet aus der Ankathete und der Gegenkathete den eingeschlossenen Winkel z. B. Alpha.

Syntax

(‹Winkel in Grad› : REAL) := **ATAN**(‹Tangenswert›)

Die Rückgabe des Winkels erfolgt in Grad.

Beispiel:

```
a     := 30              // Gegenkathete
alpha := 23.5            // Winkel Alpha
b     := a / TAN(alpha)  // Berechnung der Ankathete
```

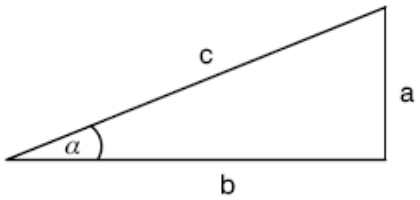
```
a      := 45.89            // Gegenkathete
b      := 145.67           // Ankathete
value  := a / b
alpha  := ATAN(value)// Berechnung des eingeschlossenen Winkels
```

17.9 Cotangens ‹COT›, ‹ACOT›

Die Cotangens-Funktion stellt den mathematischen Zusammenhang zwischen einem Winkel und den Seiten in einem rechtwinkligen Dreieck her.

Für Cotangens gilt:

Cotan(alpha) = Ankathete / Gegenkathete



cotan(alpha) = b / a

Die Angabe des Winkels erfolgt in der Einheit Grad.

Syntax

(‹Cotangenswert› : REAL) := **COT**(‹Winkel›)

Verwendung:

Die Cotangens-Funktion dient zur Berechnung von unbekannten Strecken in einem rechtwinkligen Dreieck. Ist der Winkel bekannt und auch entweder die Ankathete oder die Gegenkathete, so lässt sich die unbekannte Strecke berechnen.

Soll jedoch der Winkel berechnet werden, so stellt die CMXR die Funktion des Arcus Cotangens bereit. Diese Funktion berechnet aus der Ankathete und der Gegenkathete den eingeschlossenen Winkel z. B. Alpha.

Syntax

(‹Winkel in Grad› : REAL) := **ACOT**(‹Cotangenswert›)

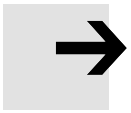
Die Rückgabe des Winkels erfolgt in Grad.

Beispiel:

```
a     := 30                 // Gegenkathete
alpha := 23.5               // Winkel Alpha
b     := a   * COT(alpha)   // Berechnung der Ankathete
a     := 45.89              // Gegenkathete
b     := 145.67             // Ankathete
value := b / a
alpha := ACOT(value)    // Berechnung eingeschlossener Winkel
```

17.10 Tangens2 ‹ATAN2›

Berechnet Arcustangens mit der Wertebereichsvergrößerung von + π bis –π.

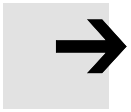


Syntax

(‹Argument› : REAL) := **ATAN2** (‹y› : REAL, ‹x› : REAL)

17.11 Logarithmus ‹LN›

Die Logarithmusfunktion berechnet den natürlichen Logarithmus des übergebenen Argumentes.

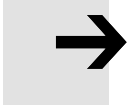


Syntax

(‹natürlicher Logarithmus› : REAL) := **LN** (‹Wert› : REAL)

17.12 Exponent ‹EXP›

Die Exponentialfunktion berechnet den Wert e(x).



Syntax

(‹Ergebnis› : REAL) := **EXP** (‹Wert›:REAL)

17.13 Absolutwert ‹ABS›

Die Funktion Absolutwert liefert den absoluten Betrag des übergebenen REAL-Wertes. Der Rückgabewert ist somit immer positiv. Negative Zahlen werden als Betrag (mit positivem Vorzeichen) zurückgegeben.

Syntax

(‹Absolutwert› : REAL) := **ABS** (‹Wert› : REAL)

Beispiel

```
Level := 452.98
Level := ABS(Level) // Level hat den absoluten Wert von 452.98
Level := -1056.61
Level := ABS(Level) // Level hat den absoluten Wert von 1056.61
```

17.14 Quadratwurzel ‹SQRT›

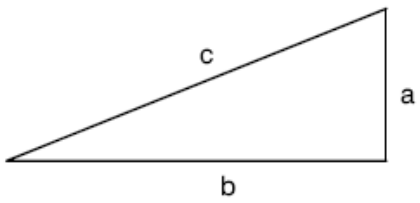
Die Wurzelfunktion ermittelt den Wert der Quadratwurzel eines Ausdrucks.

Syntax

(‹Wurzelwert› : REAL) := **SQRT** (‹Wert› : REAL)

Beispiel:

Berechnung der Länge c in einem rechtwinkligen Dreieck.



Nach dem Satz des Pythagoras gilt:

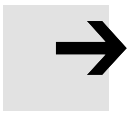
$C^2 = A^2 + B^2$ oder $C = \sqrt{(A^2 + B^2)}$

Programmbeispiel:

```
a     := 152.67           // Länge der Kathete A
b     := 63.12            // Länge der Kathete B
value := a * a + b * b
c     := SQRT(value)      // Berechnung der Hypotenuse
```

17.15 Bitweise schieben ‹SHR›, ‹SHL›

Mit den Funktionen **SHR** und **SHL** können Daten vom Typ DWORD bitweise nach rechts (Funktion **SHR**) und bitweise nach links (Funktion **SHL**) geschoben werden. Bits, die auf der einen Seite aus der Variable „herausfallen“, gehen verloren. Auf der anderen Seite werden Bits mit dem Wert "0" aufgefüllt.



Syntax

(‹Variable› :DWORD) := **SHR** (‹Wert : DWORD›, ‹count› : DINT)

(‹Variable› :DWORD) := **SHL** (‹Wert : DWORD›, ‹count› : DINT)

Beispiel:

Mit der Anweisung

```
Mask := SHL(Mask, 3)
```

wird der Inhalt der Variable Mask um 3 Bit nach links geschoben. Auf der rechten Seite werden 3 Bit mit dem Wert "0" eingeschoben.

Inhalt der Variable:

1001 1111 0000 1100 1111 1111 1001 1100

Es ergibt sich das Bitmuster:

1111 1000 0110 0111 1111 1100 1110 0**000**

Mit der Anweisung

```
Mask := SHR(Mask, 3)
```

wird der Inhalt der Variable Mask um 3 Bit nach rechts geschoben. Auf der linken Seite werden 3 Bit mit dem Wert 0 eingeschoben.

Inhalt der Variable:

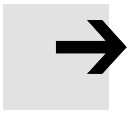
1001 1111 0000 1100 1111 1111 1001 1**100**

Es ergibt sich das Bitmuster:

0001 0011 1110 0001 1001 1111 1111 0011

17.16 Bitweise rotieren ‹ROR›, ‹ROL›

Mit den Funktionen **ROR** und **ROL** können Daten vom Typ DWORD bitweise nach rechts (Funktion **ROR**) und bitweise nach links (Funktion **ROL**) rotiert werden. Beim Rotieren des Bitmusters wird das heraus fallende Bit auf der anderen Seite wieder eingefügt.



Syntax

(‹Variable› : DWORD) := **ROR** (‹Wert : DWORD›, ‹count› : DINT)

(‹Variable› : DWORD) := **ROL** (‹Wert : DWORD›, ‹count› : DINT)

Beispiel:

Mit der Anweisung

```
Mask := ROL(Mask, 4)
```

wird der Inhalt der Variable Mask um 4 Bit nach links rotiert. Auf der rechten Seite werden die 4 Bits wieder eingefügt.

Inhalt der Variable:

1011 1111 0110 1001 0011 000 1001 1010

Es ergibt sich das Bitmuster:

1111 0110 1001 0011 000 1001 1010 **1011**

Mit der Anweisung

```
Mask := ROR(Mask, 4)
```

wird der Inhalt der Variable Mask um 4 Bit nach rechts rotiert. Auf der linken Seite werden die 4 Bits wieder eingefügt.

Inhalt der Variable:

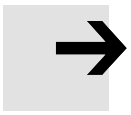
1011 1111 0110 1001 0011 000 1001 **1010**

Es ergibt sich das Bitmuster:

1010 1011 1111 0110 1001 0011 000 1001

17.17 Wert nach String wandeln ‹STR›

Wandelt einen beliebigen Datentyp in einen String um.



Syntax

(‹Variable› : STRING) := **STR** (‹Wert› : ANY)

Beispiel:

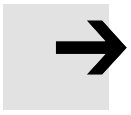
Mit der Anweisung STR wird der Wert der Variable Real1 in einen String gewandelt.

```
Real1 := 25.5
String1 := STR(Real1)
```

String1 hat nun den Wert „25.500000“.

17.18 ASCII Wert in Zeichen wandeln ‹CHR›

Wandelt einen ASCII Wert in ein einzelnes Zeichen um.



Syntax

(‹Variable› : STRING) := **CHR** (‹Wert› : DINT / DWORD)

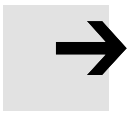
Beispiel:

Mit der folgenden Anweisung enthält die Variable String1 den Wert „a“.

```
String1 := CHR(97)
```

17.19 Zeichen in ASCII Wert wandeln ‹ORD›

Wandelt ein einzelnes Zeichnen in den ASCII Wert um.



Syntax

(‹Variable› : DINT / DWORD) := **ORD** (‹Wert› : STRING)

Beispiel:

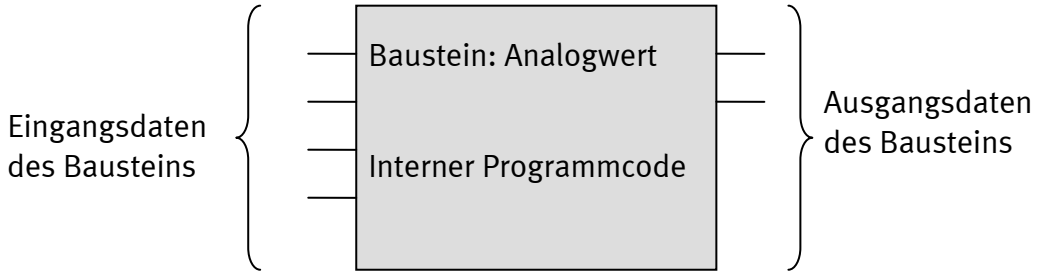
Mit der folgenden Anweisung enthält die Variable Dint1 den Wert 65.

```
Dint1 := ORD("A")
```

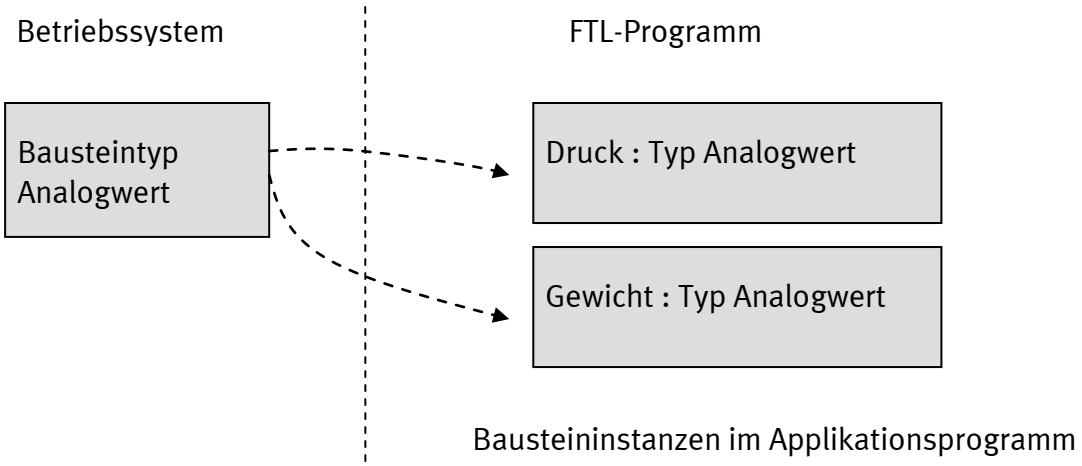
18. Bausteine

Die FTL Sprache verfügt über die Möglichkeit, Bausteine zu verwenden. Bausteine besitzen neben Daten auch Bestandteile in Form von Programmcode, um eine bestimmte Funktionalität zu erzielen. Um mit einem Baustein in FTL zu kommunizieren, besitzt dieser Ein- und Ausgangsdaten.

Grafische Darstellung eines Bausteins:



Die Anzahl der Ein- / Ausgangsdaten eines Bausteins, sind in FTL in einem Datentyp hinterlegt. Um nun einen Baustein in der Applikation zu benützen, muss dieser deklariert werden. Bei einem Baustein spricht man von einer Instanzierung. Dabei wird unter dem frei wählbaren Bausteinnamen (Bezeichner) eine Kopie der Bausteinhülle angelegt. Die Bausteinhülle umfasst dabei die Ein- und Ausgangsdaten sowie interne Programmdaten. Der eigentliche Programmcode existiert nur einmal.



Deklaration einer Bausteininstanz

‹Instanzname› **:** ‹Instanztyp› (Parameter)

Gleich wie Variablen, werden die Instanzen in der zugehörigen Datendatei abgespeichert.

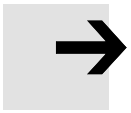
Beispiel einer Instanz

```
Sensor : AIN(...)
```

Je nach Bausteintyp werden diesem bei der Deklaration Daten mitgegeben z. B. der Name eines Hardwaresignals. Die Dokumentation der Bausteine ist den folgenden Kapiteln zu entnehmen.

18.1 Funktionen

Um auf Daten eines Bausteines zuzugreifen oder Funktionalitäten in Anspruch zu nehmen, werden Bausteinfunktionen angeboten. Der Zugriff auf die Funktionen des Bausteins erfolgt über den Instanznamen und den Punkt-Operator.



Zugriff auf Bausteinfunktionen

‹Instanzname› . ‹Funktionsname›

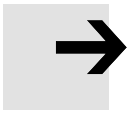
Im Beispiel wird auf die Bausteinfunktion Read der Instanz Sensor zugegriffen:

```
merker := Sensor.Read()
```

Bei manchen Funktionen können Parameter übergeben werden. Näheres ist in den folgenden Kapiteln der einzelnen Bausteine beschrieben.

18.2 Variable

Neben Funktionen können Bausteine auch Variable enthalten. Mit diesen Bausteinvariablen kann auf Daten des Bausteins zugegriffen werden. Der Zugriff auf die Variablen des Bausteins erfolgt über den Instanznamen und den Punkt-Operator.



Zugriff auf Bausteinvariable

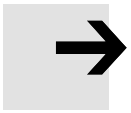
‹Instanzname› . ‹Variablenname›

Der Zugriff auf die Daten kann bei bestimmten Bausteinen über Variable und Funktionen erfolgen, sie liefern das gleiche Ergebnis. Beide Zugriffsarten haben unterschiedliche Laufzeiten. Näheres ist im Kapitel der einzelnen Bausteine beschrieben.

Im Beispiel wird auf die Bausteinvariable State der Instanz Sensor zugegriffen:

```
IF Sensor.state THEN
  Lin(pos1)
ELSE
```

```
  Lin(pos2)
END_IF
```



Empfehlung:

Wenn der Zugriff über eine Variable erfolgen kann, sollte diese Variante bevorzugt werden, sie benötigt weniger Laufzeit.

18.3 Laufzeitverhalten

Das FTL-Programm läuft nicht zyklisch, sondern wird sequentiell Anweisung für Anweisung abgearbeitet. Dieses Verhalten ist jedoch für einige Applikationen nicht ausreichend. Soll z. B. eine Prüfung parallel zum aktiven Bewegungsprogramm vorgenommen werden, ob es an einem Eingang einen Flankenwechsel gegeben hat, so müsste dies aufwendig in einem parallelen Programm gemacht werden. Um nun ein Bearbeiten eines Bausteins optimal und komfortabel zu gestalten, laufen einige Bausteine im Betriebssystem zyklisch. Der Anwender kann dann mit den Methoden oder Variablen auf die Ergebnisse des Bausteins zugreifen.

18.3.1 Parameter ovlEnable

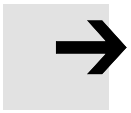
Einige Methoden der folgenden Bausteine besitzen einen optionalen Parameter **ovlEnable**. Dieser Parameter regelt den Abfragezeitpunkt des digitalen Eingangssignals. Der Parameter ist vom Typ BOOL, ihm kann der Zustand TRUE oder FALSE übergeben werden.

Zustand FALSE

Der Satzvorlauf wird an dieser Stelle gestoppt. Erreicht der Satzhauptlauf diese Programmanweisung, dann wird das Signal ausgelesen bzw. ausgeführt. Die Bewegung wird gestoppt und nach dem Befehl wieder beschleunigt. Ein Überschleifen auf folgende Bewegungen ist nicht möglich.

Zustand TRUE

Wird dem Parameter ovlEnable der Zustand TRUE übergeben, so wird zum spätesten Zeitpunkt das Signal ausgelesen bzw. ausgeführt, bei dem die Bewegung noch fortgeführt werden kann. Die Bewegung wird nicht gestoppt, sondern kontinuierlich fortgeführt. Ein Überschleifen auf folgende Bewegungen ist möglich.



Hinweis

Wird der Parameter nicht angegeben, so wird FALSE angenommen.

18.4 Digitaler Eingangsbaustein DIN

Der Baustein DIN dient zur Abfrage und Verwaltung eines digitalen Eingangs.

18.4.1 Instanzierung

Zur Instanzierung des digitalen Eingangsbausteins wird der Hardwarebezug zum digitalen Eingang benötigt. Dieser wird über den Parameter ‹input› bei der Instanzierung angegeben.

Syntax

‹Instanzname› **.DIN(** ‹Timeout› : DINT,
‹RisingEdge› : BOOL,
MAPX(“‹ input›**“)**)

Es können weitere Parameter angegeben werden, um z. B. Voreinstellungen zu treffen. Dies ist jedoch zur Instanzierung nicht notwendig und kann im Programm erfolgen.

Beispiel:

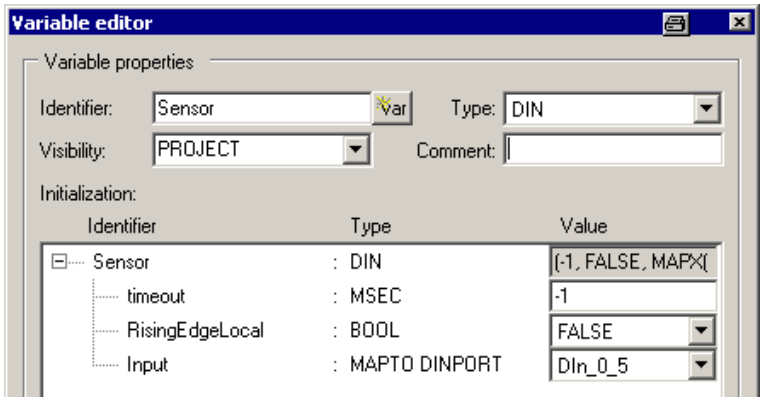
Ein Sensor wird als Bausteininstanz angelegt:

Instanzierung in der Datendatei:

```
Sensor : DIN := (-1, FALSE, MAPX("_system.Input1"))
```

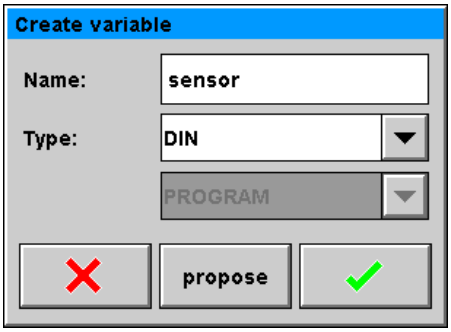
Instanzierung mit dem FTL-Editor

Die Instanzierung erfolgt im Dialog zum Anlagen von Variablen durch Auswahl des ‹Typ:› (DIN) und Eingabe des ‹Bezeichner:› (Sensor). In der Zeile ‹MAPTO DINPORT› wird der reale Bezug zu einen konfigurierten Digitalen Eingang hergestellt.



Instanzierung mit dem Handbediengerät

Die Instanzierung des digitalen Eingangsbausteins mit dem Handbediengerät erfolgt in der Variablenmaske durch Auswahl des Datentyps DIN. Der Eintrag wird automatisch im Datenfile gespeichert.



Der digitale Eingangsbaustein besitzt folgende Variablen:

Variable	Typ	Bedeutung	Zustand beim Anlegen
Timeout	REAL	Timeout zum Warten bei den Befehlen Wait und WaitN. -1 = kein Timeout (unendlich warten) 0 = sofortige Überprüfung ›0 = Timeout in Millisekunden	-1
RisingEdge	BOOL	Zustand der steigenden Flanke	FALSE
Input	DINPORT	Verweis auf den Hardwareeingang	---

Tabelle 18.1 Variablen digitaler Eingangsbaustein

Timeout

Mit dem Parameter Timeout kann eine Wartezeit definiert werden. Die Methode des Bausteins wartet die eingestellte Zeit, auf den angeforderten Signalzustand.

- Steht der Wert auf -1, so erfolgt ein unendliches Warten.
- Wird ein Wert › 0 angegeben, so wird max. diese Zeit abgewartet. Ist die Wartezeit abgelaufen, ohne dass der gewünschte Signalzustand eingetreten ist, wird der Programmablauf fortgesetzt. Eine Fehlermeldung muss vom Ablaufprogramm selbst erzeugt werden.
- Wird als Wert 0 angegeben, so erfolgt eine sofortige Prüfung. Falls das Signal nicht den gewünschten Wert besitzt, muss vom Ablaufprogramm eine Fehlermeldung generiert werden.

Weitere Informationen und ein Beispiel zum erzeugen der Fehlermeldung sind im Kapitel 20.2 Arbeiten mit Ein- und Ausgangsbausteinen beschrieben.

RisingEdge

Obwohl der Baustein nicht aufgerufen wird, speichert er den Zustand einer steigenden Flanke. Dieser Zustand kann im Programm ausgewertet werden.

Input

Die Variable Input beschreibt den Verweis auf den Hardwareeingang des Bausteins.

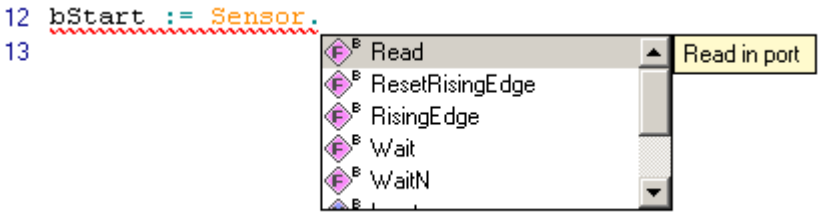
18.4.2 Methoden

Der Baustein verfügt über mehrere Methoden:

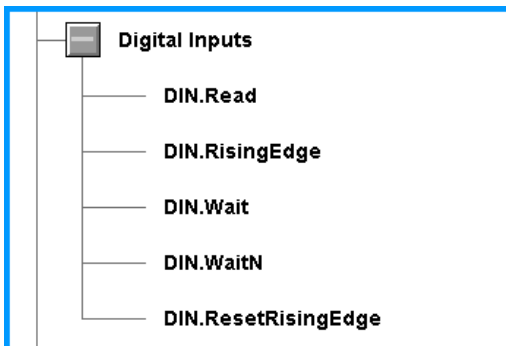
Methode	Beschreibung
Wait	warten, bis Eingang Zustand TRUE besitzt
WaitN	warten, bis Eingang Zustand FALSE besitzt
Read	Zustand des Eingangs lesen
RisingEdge	Zustand steigende Flanke lesen
ResetRisingEdge	Zustand steigende Flanke rücksetzen

Tabelle 18.2 Methoden digitaler Eingangsbaustein DIN

Im FTL-Editor werden die verfügbaren Methoden nach Eingabe des Punktes automatisch in einer Liste angezeigt und kann dort ausgewählt werden.



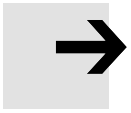
Über das Handbediengerät können diese Methoden über den Menübaum aus der Rubrik „Digitale Eingänge“ ausgewählt werden.



Die Abbildung zeigt die Auswahl der Methoden des Bausteins DIN am Handbediengerät.

18.4.3 Auf Zustand warten, Methode Wait / WaitN

Mit den Methoden Wait und WaitN kann im FTL-Programm gewartet werden, bis der Eingang den Wert TRUE oder FALSE besitzt. Der eventuell gesetzte Timeout wird dabei berücksichtigt.



Syntax

‹Instanzname›**.Wait** (OPT ovlEnable : BOOL)

‹Instanzname›**.WaitN** (OPT ovlEnable : BOOL)

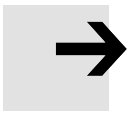
Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Beispiel:

```
Ptp(pos0)
sensor.Wait()         // warten Signal TRUE, Bewegung gestoppt
Ptp(pos1)
sensor.WaitN(TRUE)    // warten Signal FALSE, kein Stop
Ptp(pos2)
```

18.4.4 Zustand Lesen Methode Read

Mit der Methode Read kann der aktuelle Zustand des Einganges gelesen werden. Als Rückgabewert wird der Zustand TRUE oder FALSE zurückgegeben.



Synatx

‹Instanzname›**.Read** (OPT ovlEnable: BOOL) : BOOL

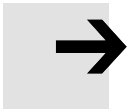
Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Beispiel:

```
value := sensor.Read()
```

18.4.5 Steigende Flanke Methode RisingEdge

Mit der Methode RisingEdge kann abgefragt werden, ob es am Eingang eine steigende Flanke gab, die gespeichert wurde. Der Rückgabewert ist auch dann TRUE, wenn der Eingang inzwischen den Zustand FALSE besitzt. Der Zustand der Flanke wird unabhängig vom Zyklus des FTL-Programms ermittelt und läuft in einem eigenen Zyklus ab.



Syntax

‹Instanzname›**.RisingEdge** (OPT ovlEnable: BOOL) : BOOL

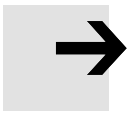
Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Beispiel:

```
value := sensor.RisingEdge()
```

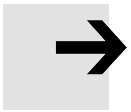
18.4.6 Flanke rücksetzen Methode ResetRisingEdge

Diese Methode setzt den internen Merker der steigenden Flanke zurück. Dies ist dann erforderlich, wenn mit der Methode RisingEdge eine steigende Flanke abgeprüft werden soll.



Syntax

‹Instanzname›.ResetRisingEdge()



Hinweis

Ist der Wert des digitalen Eingangs zum Zeitpunkt der Ausführung dieses Befehls gesetzt, so wird der Flankenerkennungszustand zurückgesetzt und nicht neuerlich gesetzt. Erst nachdem abermals eine positive Flanke (Zustandsänderung des Eingangs von 0 auf 1) auf dem Eingang aufgetreten ist, wird der Flankenerkennungszustand wieder gesetzt.

Beispiel:

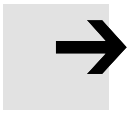
```
sensor.ResetRisingEdge()
value := sensor.RisingEgde()
```

18.5 Digitaler Ausgangsbaustein DOUT

Der Baustein DOUT dient zur Abfrage und Verwaltung eines digitalen Ausganges.

18.5.1 Instanzierung

Zur Instanzierung des digitalen Ausgangsbausteins wird der Hardwarebezug zum digitalen Ausgang benötigt. Dieser wird über den Parameter ‹output› bei der Instanzierung angegeben.



Syntax

‹Instanzname› **.DOUT(** ‹Timeout› : DINT**,**
‹RisingEdge› : BOOL,
MAPX(“‹ output›**“))**

Es können weitere Parameter angegeben werden, um z. B. Voreinstellungen zu treffen. Dies ist jedoch zur Instanzierung nicht notwendig und kann im Programm erfolgen.

Beispiel:

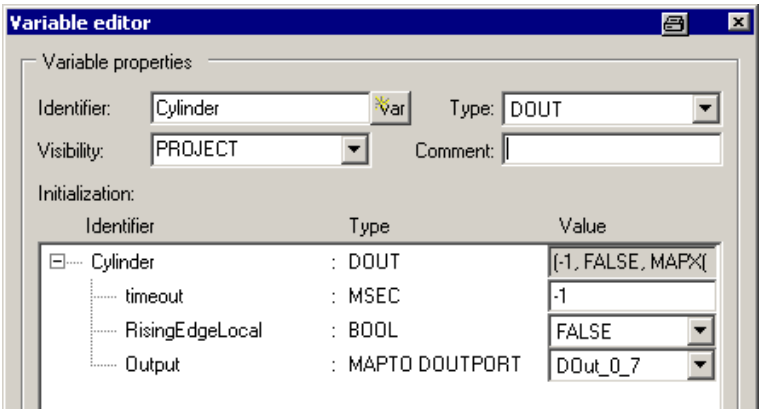
Ein Zylinder wird als Bausteininstanz angelegt:

Instanzierung in der Datendatei:

```
cylinder : DOUT := (-1,FALSE, MAPX("_system.Output3"))
```

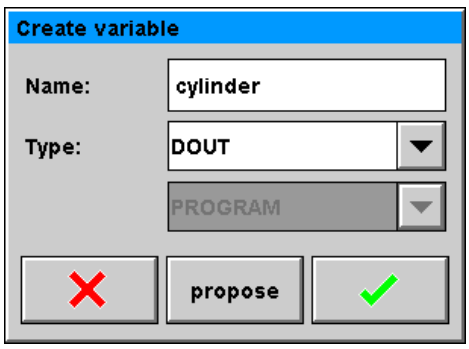
Instanzierung mit dem FTL-Editor

Die Instantiierung erfolgt im Dialog zum Anlagen von Variablen durch Auswahl des ‹Typ:› (DOUT) und Eingabe des ‹Bezeichner:› (Cylinder). In der Zeile ‹MAPTO DOUTPORT› wird der reale Bezug zu einen konfigurierten Digitalen Ausgang hergestellt.



Instanzierung mit dem Handbediengerät

Die Instanzierung des digitalen Ausgangsbausteins mit dem Handbediengerät erfolgt in der Variablenmaske durch Auswahl des Datentyps DOUT.



Mit der Instanzierung am Handbediengerät wird der Eintrag im Datenfile automatisch gespeichert.

18.5.2 Variable

Der digitale Ausgangsbaustein besitzt folgende Variablen:

Variable	Typ	Bedeutung	Zustand beim Anlegen
Timeout	REAL	Timeout zum Warten bei den Befehlen Wait und WaitN. -1 = kein Timeout (unendlich warten) 0 = sofortige Überprüfung ›0 = Timeout in Millisekunden	-1
RisingEdge	BOOL	Zustand der steigenden Flanke	FALSE
Output	DOUTPORT	Verweis auf den Hardwareausgang	---

Tabelle 18.3 Variablen digitaler Ausgangsbaustein

Timeout

Mit dem Parameter Timeout kann eine Wartezeit definiert werden. Die Methode des Bausteins wartet die eingestellte Zeit, auf den angeforderten Signalzustand.

- Steht der Wert auf -1, so erfolgt ein unendliches Warten.
- Wird ein Wert › 0 angegeben, so wird max. diese Zeit abgewartet. Ist die Wartezeit abgelaufen, ohne dass der gewünschte Signalzustand eingetreten ist, wird der Programmablauf fortgesetzt. Eine Fehlermeldung muss vom Ablaufprogramm selbst erzeugt werden.
- Wird als Wert 0 angegeben, so erfolgt eine sofortige Prüfung. Falls das Signal nicht den gewünschten Wert besitzt, muss vom Ablaufprogramm eine Fehlermeldung generiert werden.

Weitere Informationen und ein Beispiel zum erzeugen der Fehlermeldung sind im Kapitel 20.2 Arbeiten mit Ein- und Ausgangsbausteinen beschrieben.

RisingEdge

Obwohl der Baustein nicht aufgerufen wird, speichert er den Zustand einer steigenden Flanke. Dieser Zustand kann im Programm ausgewertet werden.

Output

Die Variable Output beschreibt den Verweis auf den Hardwareausgang des Bausteins.

18.5.3 Methoden

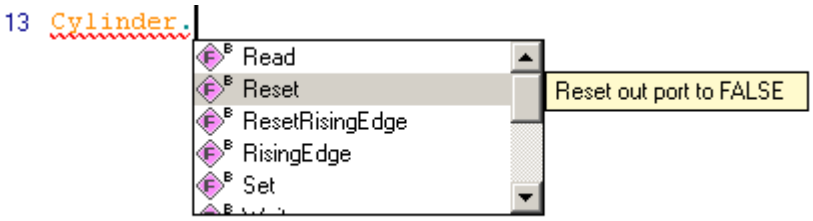
Der Baustein verfügt über mehrere Methoden:

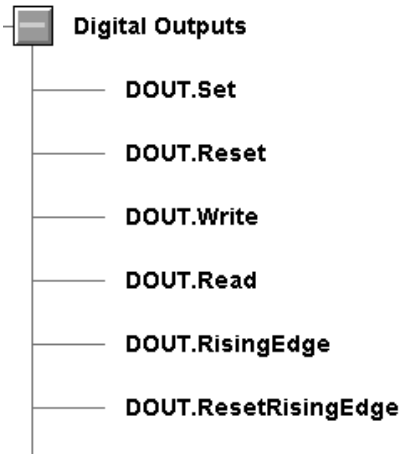
Methode	Beschreibung
Wait	Warten, bis Ausgang den Wert TRUE besitzt
WaitN	Warten, bis Ausgang den Wert FALSE besitzt
Read	Ausgangswert lesen
RisingEdge	Zustand steigende Flanke lesen
ResetRisingEdge	Zustand steigende Flanke rücksetzen
Set	Ausgang setzen, Zustand TRUE
Reset	Ausgang rücksetzen, Zustand FALSE
Write	Ausgang auf angegebenen Wert setzen

Tabelle 18.4 Methoden digitaler Ausgangsbaustein DOUT

Über das Handbediengerät können diese Methoden über den Menübaum aus der Rubrik „Digitale Ausgänge“ ausgewählt werden.

Im FTL-Editor werden die verfügbaren Methoden nach Eingabe des Punktes automatisch in einer Liste angezeigt und können dort ausgewählt werden.

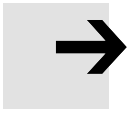




Die Abbildung zeigt die Auswahl von Methoden des Bausteins DOUT am Handbediengerät.

18.5.4 Auf Zustand warten, Methode Wait / WaitN

Mit den Methoden **Wait** und **WaitN** kann im FTL-Programm gewartet werden, bis der Ausgang den Wert TRUE oder FALSE besitzt. Der eventuell gesetzte Timeout wird dabei berücksichtigt.



Syntax

‹Instanzname›**.Wait** (OPT ovlEnable : BOOL)

‹Instanzname›**.WaitN** (OPT ovlEnable : BOOL)

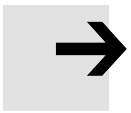
Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Beispiel:

```
Ptp(pos0)
cylinder.Wait()      // warten Ausgang TRUE, Bewegung gestoppt
Ptp(pos1)
cyinder.WaitN(TRUE) // warten Ausgang FALSE, kein Stop
Ptp(pos2)
```

18.5.5 Zustand Lesen Methode Read

Mit der Methode **Read** kann der aktuelle Zustand des Ausganges gelesen werden. Als Rückgabewert wird der Zustand TRUE oder FALSE zurückgegeben.



Syntax

‹Instanzname›**.Read** (OPT ovlEnable: BOOL) : BOOL

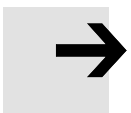
Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Beispiel:

```
value := cylinder.Read()
```

18.5.6 Steigende Flanke, Methode RisingEdge

Mit der Methode **RisingEdge** kann abgefragt werden, ob es am Ausgang eine steigende Flanke gab, die gespeichert wurde. Der Rückgabewert ist auch dann TRUE, wenn der Ausgang inzwischen den Zustand FALSE besitzt. Der Zustand der Flanke wird unabhängig vom Zyklus des FTL-Programms ermittelt und läuft in einem eigenen Zyklus ab.



Syntax

‹Instanzname›**.RisingEdge** (OPT ovlEnable: BOOL) : BOOL

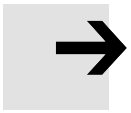
Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Beispiel:

```
value := cylinder.RisingEdge()
```

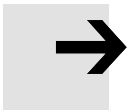
18.5.7 Flanke rücksetzen, ResetRisingEdge

Diese Methode setzt den internen Merker der steigenden Flanke zurück. Dies ist dann erforderlich, wenn mit der Methode **RisingEdge** eine steigende Flanke abgeprüft werden soll.



Syntax

‹Instanzname›.**ResetRisingEdge**()



Hinweis

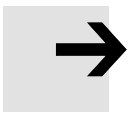
Ist der Wert des digitalen Ausgangs zum Zeitpunkt der Ausführung dieses Befehls gesetzt, so wird der Flankenerkennungszustand zurückgesetzt und nicht neuerlich gesetzt. Erst nachdem abermals eine positive Flanke (Zustandsänderung des Ausgangs von 0 auf 1) auf dem Ausgang aufgetreten ist, wird der Flankenerkennungszustand wieder gesetzt.

Beispiel:

```
cylinder.ResetRisingEdge()
value := cylinder.RisingEgde()
```

18.5.8 Setzen und rücksetzen, Methode Set / Reset

Mit den Methoden **Set** wird ein Ausgang gesetzt (Zustand TRUE) und mit **Reset** zurückgesetzt (Zustand FALSE).



Syntax

‹Instanzname›.**Set** ()

‹Instanzname›.**Reset** ()

Set und Reset werden im Hauptlauf ausgeführt. Das Überschleifen wird durch den Baustein NICHT beeinflusst.

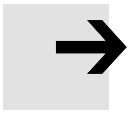
Beispiel:

```
Lin(pos1)
Lin(pos2)
```

```
cylinder.Set()       // setzt Ausgang cylinder auf TRUE
Lin(pos3)
cylinder.Reset()     // setzt Ausgang cylinder auf FALSE
Lin(pos4)
```

18.5.9 Setzen des Ausgangs, Methode Write

Mit der Methode **Write** kann der Zustand eines Ausgangs beschrieben werden. Der gewünschte Zustand TRUE oder FALSE wird dabei als Parameter übergeben. Die Funktionsweise von **Write** ist die gleiche wie bei den Methoden **Set** und **Reset**.



Syntax

‹Instanzname›.**Write** (Value: BOOL)

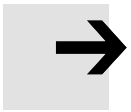
Die Methode **Write** wird im Hauptlauf ausgeführt. Das Überschleifen wird durch den Baustein NICHT beeinflusst.

Beispiel:

```
cylinder.Write(TRUE)      // setzt Ausgang cylinder auf TRUE
state := FALSE
cylinder.Write(state)     // setzt Ausgang cylinder auf FALSE
```

18.5.10 Setzen des Ausgangs auf bestimmte Dauer, Methode Pulse

Die Methode **Pulse** setzt einen Ausgang für eine bestimmte Dauer auf TRUE. Die Ausführung von **Pulse** erfolgt im Hauptlauf und beeinflusst das Überschleifen nicht.



Syntax

‹Instanzname›.**Pulse** (timeMs :DINT;
OPT pauseAtInterrupt: BOOL)

Parameter	Typ	Bedeutung
timeMs	DINT	Zeit für die Impulslänge in msec
pauseAtInterrupt	BOOL	Anhalten der Impulszeit, wenn Programm gestoppt

Tabelle 18.5 Parameter Methode Pulse

Die Methode Pulse steuert einen digitalen Ausgang mit einem Puls bestimmter Länge an. Die Länge des Pulses wird dabei in Millisekunden angegeben werden. Dabei wird zu Beginn des Pulses der Ausgang immer auf TRUE gesetzt und am Ende immer auf FALSE gesetzt. Besitzt der Ausgang jedoch schon den Zustand TRUE, so wird dieser nach Ablauf der Impulszeit zurückgesetzt.

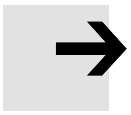
Mit dem optionalen Parameter „pauseAtInterrupt“ kann angegeben werden, wie die Reaktion im Falle eines Stopp des FTL Programm erfolgt. Wir dieser Parameter nicht angegeben, so wird intern der Zustand FALSE gesetzt.

pauseAtInterruppt = TRUE:

Mit TRUE wird der Impulszeit angehalten und der Ausgang auf den Wert FALSE gesetzt. Wird das FTL Programm fortgesetzt, so wie der Ausgang für die restliche verbleibende Zeit erneut auf TRUE gesetzt. Nach Ablauf der Impulszeit erfolgt ein Setzen auf den Zustand FALSE.

pauseAtInterruppt = FALSE:

Ist der optionale Parameter nicht oder FALSE angegeben, so wird der Impuls nach Start bis zum Ende ausgeführt. Ein Stopp des FTL Programm hat keine Auswirkungen.



Hinweis

Wird ein gestopptes FTL Programm entladen, so wird Impulsfunktion automatisch beendet und der Ausgang FALSE gesetzt.

Beispiel:

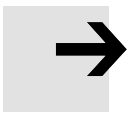
```
// setzt Ausgang cylinder für 200msec auf TRUE
cylinder.Pulse(200)
```

18.6 Analoger Eingangsbaustein AIN

Der Baustein AIN dient zur Abfrage und Verwaltung eines analogen Einganges.

18.6.1 Instanzierung

Zur Instanzierung des analogen Eingangsbausteins wird der Hardwarebezug zum analogen Eingang benötigt. Dieser wird über den Parameter Input bei der Instanzierung angegeben.



Syntax

‹Instanzname› **.AIN(** ‹Timeout› : DINT**, MAPX(“**‹ input›**“)**)

Der Parameter input ist der Name, mit dem der Hardwareeingang bezeichnet ist. Diese Zuweisung stellt einen Bezug des Bausteins zur Handware her. Es können weitere Parameter angegeben werden, um z. B. Voreinstellungen zu treffen. Dies ist jedoch zur Instanzierung nicht notwendig und kann im Programm erfolgen.

Beispiel:

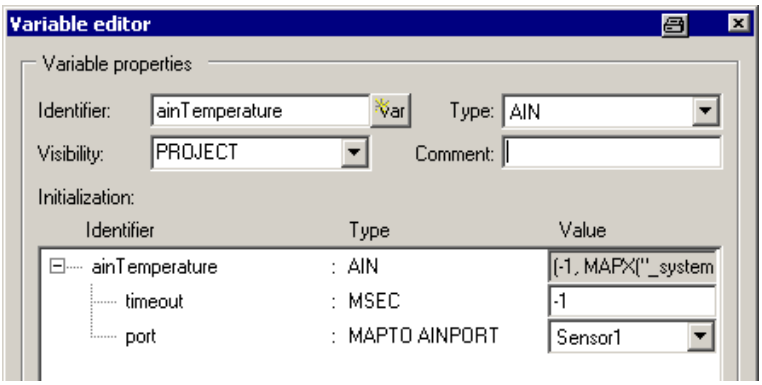
Ein Sensor wird als Bausteininstanz angelegt:

Instanzierung in der Datendatei:

```
temperature : AIN := (-1, MAPX("_system.Sensor1"))
```

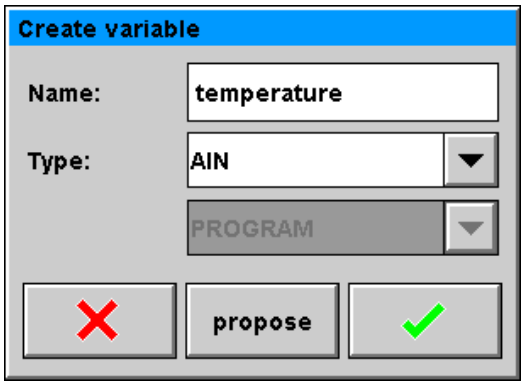
Instanzierung mit dem FTL-Editor

Die Instantiierung erfolgt im Dialog zum Anlagen von Variablen durch Auswahl des ‹Typ:› (AIN) und Eingabe des ‹Bezeichner:› (ainTemperature). In der Zeile ‹MAPTO AINPORT› wird der reale Bezug zu einen konfigurierten Analogen Eingang hergestellt.



Instanzierung mit dem Handbediengerät

Die Instanzierung des analogen Eingangsbausteins mit dem Handbediengerät erfolgt in der Variablenmaske durch Auswahl des Datentyps AIN.



Mit der Instanzierung am Handbediengerät wird der Eintrag im Datenfile automatisch gespeichert.

18.6.2 Variable

Der analoge Eingangsbaustein besitzt folgende Variablen:

Variable	Typ	Bedeutung	Zustand beim Anlegen
Timeout	REAL	Timeout zum Warten bei den Befehlen WaitLss, WaitGrt, WaitIns und Waitouts. -1 = kein Timeout (unendlich warten) 0 = sofortige Überprüfung ›0 = Timeout in Millisekunden	-1
Input	AINPORT	Verweis auf den Hardwareeingang	---

Tabelle 18.6 Variablen analoger Eingangsbaustein

Timeout

Mit dem Parameter Timeout kann eine Wartezeit definiert werden. Die Methode des Bausteins wartet die eingestellte Zeit, auf den angeforderten Signalzustand.

- Steht der Wert auf -1, so erfolgt ein unendliches Warten.
- Wird ein Wert › 0 angegeben, so wird max. diese Zeit abgewartet. Ist die Wartezeit abgelaufen, ohne dass der gewünschte Signalzustand eingetreten ist, wird der Programmablauf fortgesetzt. Eine Fehlermeldung muss vom Ablaufprogramm selbst erzeugt werden.
- Wird als Wert 0 angegeben, so erfolgt eine sofortige Prüfung. Falls das Signal nicht den gewünschten Wert besitzt, muss vom Ablaufprogramm eine Fehlermeldung generiert werden.

Weitere Informationen und ein Beispiel zum erzeugen der Fehlermeldung sind im Kapitel 20.2 Arbeiten mit Ein- und Ausgangsbausteinen beschrieben.

Input

Die Variable Input beschreibt den Verweis auf den Hardwareeingang des Bausteins.

18.6.3 Methoden

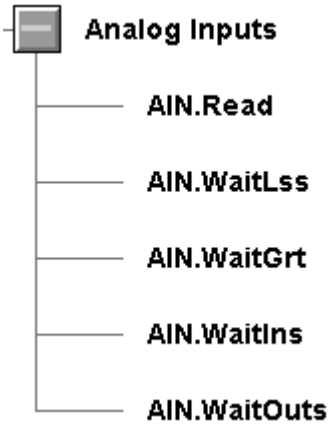
Der Baustein verfügt über mehrere Methoden:

Methode	Beschreibung
WaitLss	Warten, bis Eingangswert kleiner als angegeben
WaitGrt	Warten bis Eingangswert größer als angegeben
WaitIns	Warten, bis Eingangswert innerhalb der angegebenen Grenze
WaitOuts	Waten, bis Eingangswert außerhalb der angegebenen Grenze
Read	Eingangswert lesen

Tabelle 18.7 Methoden analoger Eingangsbaustein

Im FTL-Editor werden die verfügbaren Methoden nach Eingabe des Punktes automatisch in einer Liste angezeigt und können dort ausgewählt werden.

Über das Handbediengerät können diese Methoden über den Menübaum aus der Rubrik „Analoge Eingänge“ ausgewählt werden.



Die Abbildung zeigt die Auswahl von Methoden des Bausteins AIN am Handbediengerät.

18.6.4 Warten bis Wert kleiner/größer, Methode WaitLss , WaitGrt

Die Methoden **WaitLss** und **WaitGrt** ermöglichen ein Abfragen eines analogen Einganges auf einen Zustand größer oder kleiner des angegebenen Wertes. Der Parameter Timeout des Bausteins wird dabei berücksichtigt.

Syntax

‹Instanzname›**.WaitLss**(value : REAL, OPT ovlEnable : BOOL)

‹Instanzname›**.WaitGrt**(value : REAL, OPT ovlEnable : BOOL)

Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Parameter	Typ	Bedeutung
value	REAL	Grenzwert

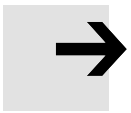
Tabelle 18.8 Parameter Methoden WaitLss, WaitGrt

Beispiel:

```
Lin(pos1)
temperature.WaitLss(65.0, TRUE)
Lin(pos2)
```

18.6.5 Warten bis Wert im/aus Bereich, Methode WaitIns, WaitOuts

Mit den Methoden **WaitIns** und **WaitOuts** kann gewartet werden, bis sich der analoge Eingangswert innerhalb oder außerhalb des angegebenen Wertebereiches befindet. Der Parameter Timeout des Bausteins wird dabei berücksichtigt.



Syntax

‹Instanzname›**.WaitIns**(minValue : REAL,
maxValue : REAL,
OPT ovlEnable : BOOL)

‹Instanzname›**.WaitOuts**(minValue : REAL,
maxValue : REAL,
OPT ovlEnable : BOOL)

Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Parameter	Typ	Bedeutung
minValue	REAL	Unterer Grenzwert
maxValue	REAL	Oberer Grenzwert

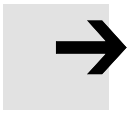
Tabelle 18.9 Parameter Methoden WaitIns, WaitOuts

Beispiel

```
Lin(pos1)
// warten bis Temperatur innerhalb der Grenzen
temperature.WaitIns(50.0, 55.0, TRUE)
Lin(pos2)
```

18.6.6 Abfragen des Wertes, Methode Read

Die Methode Read liest den aktuellen Wert des analogen Einganges ein.



Syntax

‹Instanzname›**.Read**(OPT ovlEnable)

Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Beispiel

```
Lin(pos1)
// lesen der Temperatur ohne Stopp auf der Bahn
value := temperature.Read(TRUE)
Lin(pos2)
```

18.7 Analoger Ausgangsbaustein AOUT

Der Baustein AOUT dient zur Abfrage und Verwaltung eines analogen Ausganges.

18.7.1 Instanzierung

Zur Instanzierung des analogen Ausgangsbausteins wird der Hardwarebezug zum analogen Ausgang benötigt. Dieser wird über den Parameter ‹port› bei der Instanzierung angegeben.

Syntax

‹Instanzname› **.AOUT(** ‹Timeout› : DINT, **MAPX(“**‹ port›**“)**)

Es können weitere Parameter angegeben werden, um z. B. Voreinstellungen zu treffen. Dies ist jedoch zur Instanzierung nicht notwendig und kann im Programm erfolgen.

Beispiel:

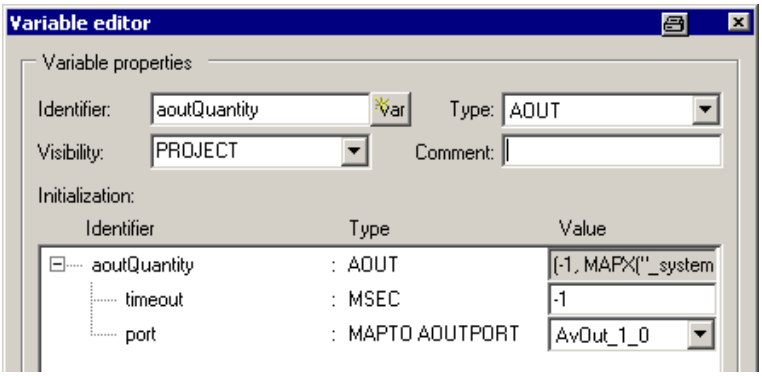
Eine Mengenvorgabe wird als Bausteininstanz angelegt:

Instanzierung in der Datendatei:

```
quantity : AOUT := (-1, MAPX("_system.SetValue"))
```

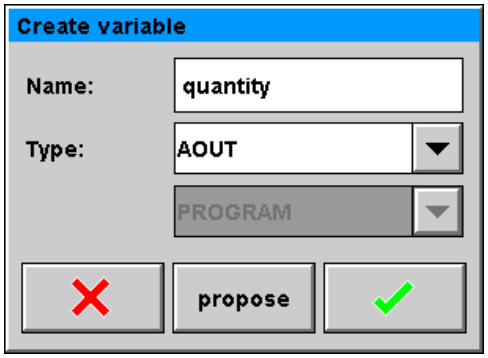
Instanzierung mit dem FTL-Editor

Die Instanzierung erfolgt im Dialog zum Anlagen von Variablen durch Auswahl des ‹Typ:› (AOUT) und Eingabe des ‹Bezeichner:› (aoutQuantity). In der Zeile ‹MAPTO AOUTPORT› wird der reale Bezug zu einen konfigurierten Analogen Ausgang hergestellt.



Instanzierung mit dem Handbediengerät

Die Instanzierung des analogen Ausgangsbausteins mit dem Handbediengerät erfolgt in der Variablenmaske durch Auswahl des Datentyps AOUT.



Mit der Instanzierung am Handbediengerät wird der Eintrag im Datenfile automatisch gespeichert.

18.7.2 Variable

Der analoge Ausgangsbaustein besitzt folgende Variablen:

Variable	Typ	Bedeutung	Zustand beim Anlegen
Timeout	REAL	Timeout zum Warten bei den Befehlen WaitLss, WaitGrt, WaitIns und Waitouts. -1 = kein Timeout (unendlich warten) 0 = sofortige Überprüfung ›0 = Timeout in Millisekunden	-1
Output	AOUTPORT	Verweis auf den Hardwareausgang	---

Tabelle 18.10 Variablen analoger Ausgangsbaustein

Timeout

Mit dem Parameter Timeout kann eine Wartezeit definiert werden. Die Methode des Bausteins wartet die eingestellte Zeit, auf den angeforderten Signalzustand.

- Steht der Wert auf -1, so erfolgt ein unendliches Warten.
- Wird ein Wert › 0 angegeben, so wird max. diese Zeit abgewartet. Ist die Wartezeit abgelaufen, ohne dass der gewünschte Signalzustand eingetreten ist, wird der Programmablauf fortgesetzt. Eine Fehlermeldung muss vom Ablaufprogramm selbst erzeugt werden.
- Wird als Wert 0 angegeben, so erfolgt eine sofortige Prüfung. Falls das Signal nicht den gewünschten Wert besitzt, muss vom Ablaufprogramm eine Fehlermeldung generiert werden.

Weitere Informationen und ein Beispiel zum erzeugen der Fehlermeldung sind im Kapitel 20.2 Arbeiten mit Ein- und Ausgangsbausteinen beschrieben.

Output

Die Variable Output beschreibt den Verweis auf den Hardwareausgang des Bausteins.

18.7.3 Methoden

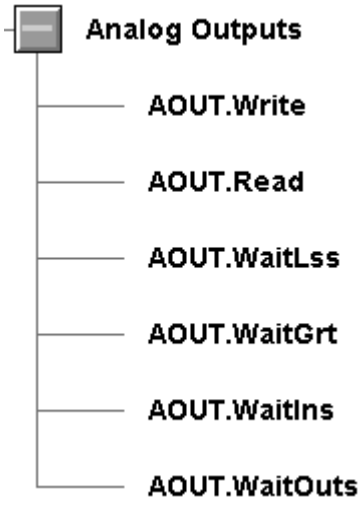
Der Baustein verfügt über mehrere Methoden:

Methode	Beschreibung
Write	Schreibt einen Wert auf den analogen Ausgang
WaitLss	Warten, bis Ausgangswert kleiner als angegeben
WaitGrt	Warten bis Ausgangswert größer als angegeben
WaitIns	Warten, bis Ausgangswert innerhalb der angegebenen Grenze
WaitOuts	Waten, bis Ausgangswert außerhalb der angegebenen Grenze
Read	Ausgangswert lesen

Tabelle 18.11 Methoden analoger Ausgangsbaustein

Im FTL-Editor werden die verfügbaren Methoden nach Eingabe des Punktes automatisch in einer Liste angezeigt und können dort ausgewählt werden.

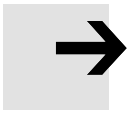
Über das Handbediengerät können diese Methoden über den Menübaum aus der Rubrik „Analoge Ausgänge“ ausgewählt werden.



Die Abbildung zeigt die Auswahl von Methoden des Bausteins AOUT am Handbediengerät.

18.7.4 Schreiben des Ausgangswertes, Methode Write

Mit der Methode **Write** wird ein analoger Ausgang geschrieben. Der gewünschte Sollwert wird in den Parameter value übergeben.



Syntax

‹Instanzname›.**Write**(value :REAL)

Parameter	Typ	Bedeutung
value	REAL	Sollwert für den analogen Ausgang

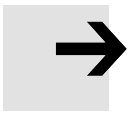
Tabelle 18.12 Parameter Methode Write

Beispiel:

```
quantity.Write(110.0)   // setzt den Ausgang auf 110.0
value := 1.3
quantity.Write(value)   // setzt den Ausgang auf 1.3
```

18.7.5 Warten bis Wert größer/kleiner, Methode WaitLss, WaitGrt

Die Methoden **WaitLss** und **WaitGrt** ermöglichen ein Abfragen eines analogen Ausgangs auf einen Zustand größer oder kleiner des angegebenen Wertes. Der Parameter Timeout des Bausteins wird dabei berücksichtigt.



Syntax

‹Instanzname›.**WaitLss**(value : REAL, OPT ovlEnable : BOOL)

‹Instanzname›.**WaitGrt**(value : REAL, OPT ovlEnable : BOOL)

Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Parameter	Typ	Bedeutung
value	REAL	Grenzwert

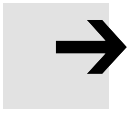
Tabelle 18.13 Parameter Methoden WaitLss, WaitGrt

Beispiel:

```
Lin(pos1)
// warten, bis weniger als 110.0
quantity.WaitLss(110.0, TRUE)
Lin(pos2)
```

18.7.6 Warten bis Wert im/aus Bereich, Methode WaitIns, WaitOuts

Mit den Methoden **WaitIns** und **WaitOuts** kann gewartet werden, bis sich der analoge Ausgangswert innerhalb oder außerhalb des angegebenen Wertebereiches befindet. Der Parameter Timeout des Bausteins wird dabei berücksichtigt.



Syntax

‹Instanzname›.**WaitIns**(minValue : REAL,
maxValue : REAL, OPT ovlEnable : BOOL)

‹Instanzname›.**WaitOuts**(minValue : REAL,
maxValue : REAL, OPT ovlEnable : BOOL)

Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Parameter	Typ	Bedeutung
minValue	REAL	Unterer Grenzwert
maxValue	REAL	Oberer Grenzwert

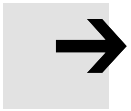
Tabelle 18.14 Parameter Methoden WaitIns, WaitOuts

Beispiel

```
Lin(pos1)
// warten bis Menge innerhalb der Grenzen
quantity.WaitIns(100.0, 110.0, TRUE)
Lin(pos2)
```

18.7.7 Ausgangswert lesen, Methode Read

Die Methode Read liest den aktuellen Wert des analogen Ausganges ein.



Syntax

‹Instanzname›.**Read**(OPT ovlEnable)

Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Beispiel

```
Lin(pos1)
// lesen der Menge ohne Stopp auf der Bahn
value := quantity.Read(TRUE)
Lin(pos2)
```

18.8 Uhrenbaustein CLOCK

Der Uhrenbaustein (Timer) kann zur einfachen Zeitmessung im Programmablauf verwendet werden. Die Zeiten werden in Millisekunden gemessen.

18.8.1 Instanzierung

Zur Instanzierung des Uhrenbausteins **CLOCK** wird der Name des Bausteins benötigt. Der Baustein besitzt keine Variablen.

Instanzierung

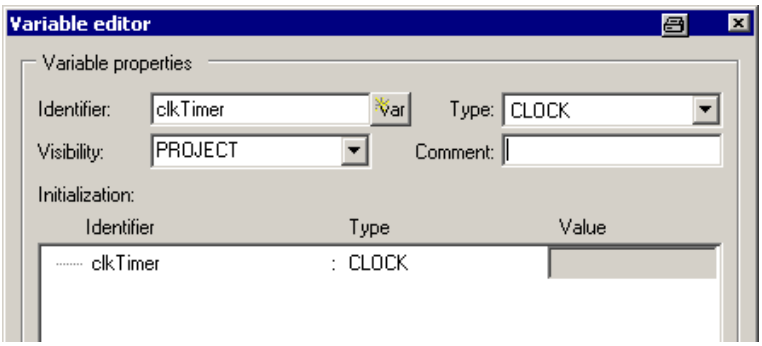
‹Instanzname› **: CLOCK**

Instanzierung in der Datendatei:

```
Timer : CLOCK
```

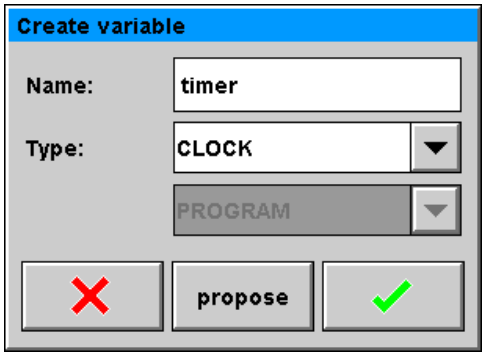
Instanzierung mit dem FTL-Editor

Die Instanziierung erfolgt im Dialog zum Anlagen von Variablen durch Auswahl des ‹Typ:› (CLOCK) und Eingabe des ‹Bezeichner:› (clkTimer).



Instanzierung mit dem Handbediengerät

Die Instanzierung des Uhrenbausteins mit dem Handbediengerät erfolgt in der Variablenmaske durch Auswahl des Datentyps CLOCK.



Mit der Instanzierung am Handbediengerät wird der Eintrag im Datenfile automatisch geschrieben.

18.8.2 Methoden

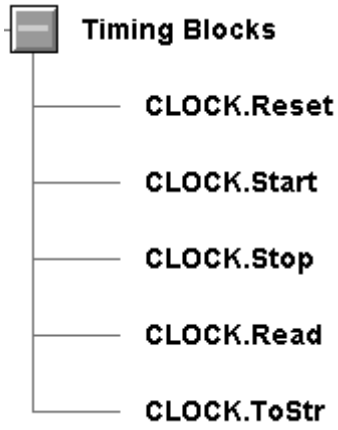
Der Uhrenbaustein verfügt über folgende Methoden:

Methode	Beschreibung
Reset	Rücksetzen der Uhr
Start	Starten der Uhr
Stop	Stoppen der Uhr
Read	Auslesen der gestoppten Zeit in msec
ToStr	Umwandeln der Zeit in einen String

Tabelle 18.15 Methoden Uhrenbaustein

Im FTL-Editor werden die verfügbaren Methoden nach Eingabe des Punktes automatisch in einer Liste angezeigt und können dort ausgewählt werden.

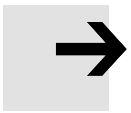
Über das Handbediengerät können diese Methoden über den Menübaum aus der Rubrik „Timerbausteine“ ausgewählt werden.



Die Abbildung zeigt die Auswahl der Methoden des Bausteins **CLOCK** am Handbediengerät.

18.8.3 Starten der Uhr, Methode Start

Mit der Methode **Star**t wird der Timer gestartet. Der Startzeitpunkt wird im Baustein intern gespeichert. Erfolgt ein Start auf einen bereits gestoppten Timer, so wird der gestoppte Timer erneut gestartet, die Zeit läuft weiter. Der Timer kann nur über den **Reset** Befehl neu gestartet werden.



Syntax

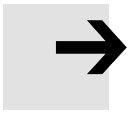
‹Instanzname›**.Start ( )**

Beispiel:

```
Timer.Start()    // Uhr wird gestartet
```

18.8.4 Stoppen der Uhr, Methode Stop

Die Methode **Stop** stoppt eine gestartete Zeitmessung. Die abgelaufene Zeit wird im Baustein intern berechnet und zum Auslesen bereitgestellt.



Syntax

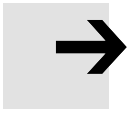
‹Instanzname›**.Stop ( )**

Beispiel:

```
Timer.Stop()    // Uhr wird angehalten
```

18.8.5 Rücksetzen der Uhr, Methode Reset

Der Aufruf der Methode **Reset** stellt den Timer zurück. Wird ein Reset auf einen laufenden Timer ausgeführt, so wird dieser auf 0 gesetzt und läuft weiter.



Syntax

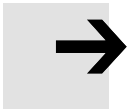
‹Instanzname›**.Reset ( )**

Beispiel:

```
Timer.Reset()    // Uhr wird rückgestellt
```

18.8.6 Auslesen der Uhr, Methode Read

Mit der Methode **Read** kann der aktuelle Zeitwert in Millisekunden (msec) ausgelesen werden.



Syntax

‹Instanzname›**.Read(**OPT ovlEnable : BOOL**)** : DINT

Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Rückgabewert: Gemessene Zeit in Millisekunden (msec)

Beispiel:

Daten:

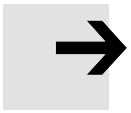
```
value      : DINT
```

Programm:

```
:
Lin(pos1)
value := Timer.Read()    // auslesen mit Bahnstopp
Lin(pos2)
value := Timer.Read(TRUE)    // auslesen im Satzhauptlauf
Lin(pos3)
:
```

18.8.7 Zeitwert in String konvertieren

Ein Aufruf der Methode **ToStr** konvertiert den aktuellen Zeitwert in einen String, der den Aufbau ‹Tage› ‹Stunden›:‹Minuten›:‹Sekunden›:‹Millisekunden› besitzt.



Syntax

‹Instanzname›**.ToStr(**OPT ovlEnable : BOOL**)** : STRING

Beschreibung des Parameters ovlEnable siehe Kapitel 18.3.1 auf Seite 158.

Rückgabewert: String mit der formatierten Zeitinformation tt hh:mm:dd.ms.

Beispiel:

Daten:

```
time  : STRING
```

Programm:

```
Lin(pos1)
```

```
time := Timer.ToStr()          // auslesen mit Bahnstopp
Lin(pos2)
time := Timer.ToStr(TRUE)      // auslesen im Satzhauptlauf
Lin(pos3)
```

18.9 Encoder-Baustein ENCODER

Der Encoder-Baustein kann im Programm verwendet werden, um die maximal zwei angeschlossenen Inkrementalgeber auszulesen.

18.9.1 Instanzierung

Zur Instanzierung des Encoder-Bausteins wird der Hardwarebezug zum Eingang benötigt. Dieser wird über den Parameter ‹port› bei der Instanzierung angegeben.

Syntax

‹Instanzname› **.ENCODER(** ‹Timeout› : DINT, **MAPX(“**‹port›**“)**)

Es können weitere Parameter angegeben werden, um z. B. Voreinstellungen zu treffen. Dies ist jedoch zur Instanzierung nicht notwendig und kann im Programm erfolgen.

Beispiel:

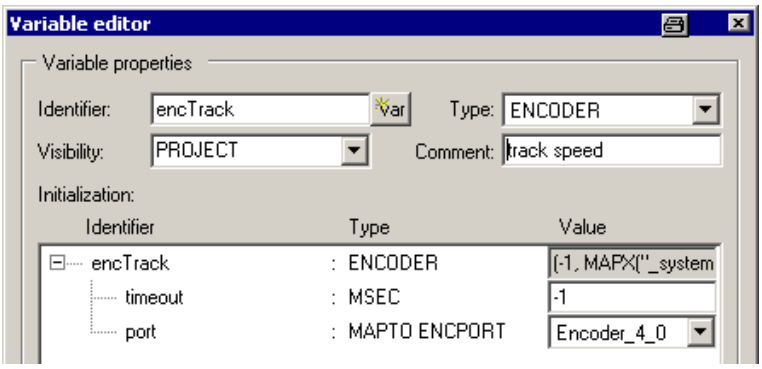
Ein Encoder wird als Bausteininstanz angelegt:

Instanzierung in der Datendatei:

```
enc0 : ENCODER := (-1, MAPX("_system.Encoder_0"))
```

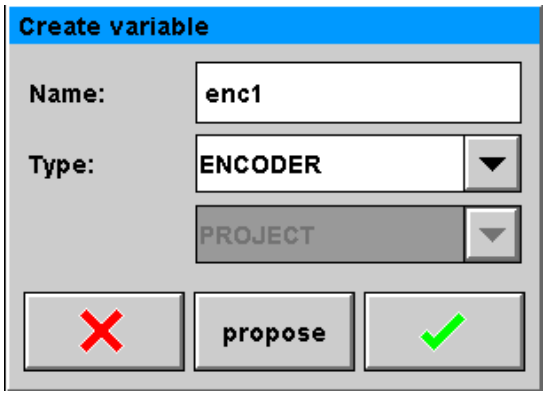
Instanzierung mit dem FTL-Editor

Die Instantiierung erfolgt im Dialog zum Anlegen von Variablen durch Auswahl des ‹Typ:› (ENCODER) und Eingabe des ‹Bezeichner:› (encTrack). In der Zeile ‹MAPTO ENCPORT› wird der reale Bezug zu einen konfigurierten Encoder-Eingang hergestellt.



Instanzierung mit dem Handbediengerät

Die Instanzierung des Encoderbausteins mit dem Handbediengerät erfolgt in der Variablenmaske durch Auswahl des Datentyps ENCODER.



Mit der Instanzierung am Handbediengerät wird der Eintrag im Datenfile automatisch gespeichert.

18.9.2 Variable

Der Encoder-Baustein besitzt folgende Variablen:

Variable	Typ	Bedeutung	Zustand beim Anlegen
timeout	REAL	Timeout zum Warten.	-1
port	ENCPORT	Verweis auf den Hardwareeingang	---

Tabelle 18.16 Variablen Encoder-Baustein

Timeout – (derzeit in diesem Baustein ohne Funktion)

Mit dem Parameter Timeout kann eine Wartezeit definiert werden. Die Methode des Bausteins wartet die eingestellte Zeit, auf den angeforderten Signalzustand. Derzeit besitzt der Encoder-Baustein keine Methoden, die den Timeout berücksichtigen.

Port

Die Variable Port beschreibt den Verweis auf den Hardwareeingang.

18.9.3 Methoden

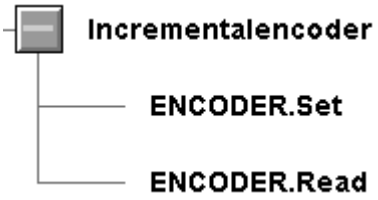
Der Encoder-Baustein verfügt über folgende Methoden:

Methode	Beschreibung
Set	Inkrementalgeberwert manuell setzen
Read	Inkrementalgeberwert auslesen

Tabelle 18.17 Methoden Encoder-Baustein

Im FTL-Editor werden die verfügbaren Methoden nach Eingabe des Punktes automatisch in einer Liste angezeigt und können dort ausgewählt werden.

Über das Handbediengerät können diese Methoden über den Menübaum aus der Rubrik „Inkrementalgeber“ ausgewählt werden.



Die Abbildung zeigt die Auswahl der Methoden des Encoder-Bausteins am Handbediengerät.

18.9.4 Encoder setzen, Methode Set

Mit der Methode **Set** kann ein Inkrementalgeberwert vom Anwenderprogramm gesetzt werden.

Syntax:

‹ENCODER›.**Set** (OPT ‹count› : DINT,
OPT ‹mode› : SETCOUNTMODE)

Parameter	Typ	Bedeutung
count	DINT	Wert auf den der Inkremetalgeber gesetzt werden soll (Inkremente)
mode	SETCOUNTMODE	Parameter zum Einstellen des Übernahmezeitpunkts

Tabelle 18.18 Parameter Methode Set

Der Baustein setzt den Wert hauptlaufsynchron, stoppt den Vorlauf aber nicht. Im Parameter mode kann der Zeitpunkt des Setzens definiert werden.

mode	Bedeutung
DIRECT	Wert wird im nächsten IO-System-Zyklus übernommen
ZEROPULSE	Wert wird beim nächsten Nullimpuls des Gebers übernommen

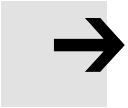
Tabelle 18.19 Modes der Methode Set

Beispiel:

```
enc0.Set(1000, DIRECT)
```

18.9.5 Encoder lesen Methode Read

Mit der Methode **Read** kann der aktuelle Inkrementalgeberwert vom Anwenderprogramm gelesen werden.



Syntax:

‹ENCODER›.**Read** (OPT ‹ovlEnable› : BOOL) : DINT

Parameter	Typ	Bedeutung
ovlEnable	BOOL	Parameter zum Einstellen des Abfragezeitpunkts

Tabelle 18.20 Parameter Methode Read

Die Methode gibt den inkrementellen Wert des Encoders zurück. Dieser wird nicht mit dem Umrechnungsfaktor verrechnet. Soll der normierte Wert gelesen werden, so muss direkt auf den Port des Encoder-Eingangs mit dem Element Value zugegriffen werden.

Beispiel normierten Wert lesen:

```
value := enc0.port.value
```

Beispiel inkrementellen Wert lesen:

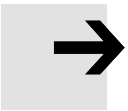
```
value := enc0.Read()
```

18.10 CANopen Baustein COPDEVICE

Der CANopen-Baustein kann im Programm verwendet werden um Geräte anzusprechen, die per Peripherie-CAN-Bus mit der CMXR verbunden sind.

Vorsicht

Am Peripherie-CAN-Bus können CANopen-Geräte angesteuert werden. Diese Geräte unterliegen jedoch nicht dem NOT-AUS-Verhalten der Kinematik. Es erfolgt kein kontrolliertes Abbremsen von Hilfsantrieben. Wird der Peripherie-CAN-Bus aus einem parallelen Programm bedient, so wird dieser im Falle eines NOT-AUS nicht gestoppt!



Hinweis

Wie einzelne Geräte angesprochen werden müssen, die per CAN-Bus mit der CMXR verbunden sind, ist der jeweiligen Herstellerdokumentation des Busteilnehmers zu entnehmen.

18.10.1 Instanzierung

Die Instanzierung des CANopen-Bausteins erfolgt aufgrund der IO-Konfiguration der Steuerung automatisch und ist manuell nicht möglich.

Der Zugriff auf angeschlossene CANopen-Geräte erfolgt über einen eindeutigen Namen, welcher über die Konfigurationssoftware FCT vorgegeben wird. Der im FCT angegebene Name kann dabei nicht direkt verwendet werden, sondern es erfolgt eine Namenserweiterung mit der Endung „_copd“. Dies gilt sowohl für Antriebe, welche am DriveBus angeschlossen sind, als auch für CANopen-Geräte am Peripherie-CAN-Bus.

Beispiel:

Die Achse 1 der Kinematik wird im FCT in der DriveBus-Konfiguration als Axis1 bezeichnet. Der Zugriff erfolgt über Axis1_copd.BausteinMethode().

CANopen-Konfiguration:

```
[IO.ONBOARD.SLOTCAN:0.FX200A:0.CAN:0.COPDEVICE:1]
name="Axis1"    // Name des COP-Device
```

Programmcode:

```
Axis1_copd.WriteSDO(16#6060,0,7,eight_bits)
```

18.10.2 Methoden

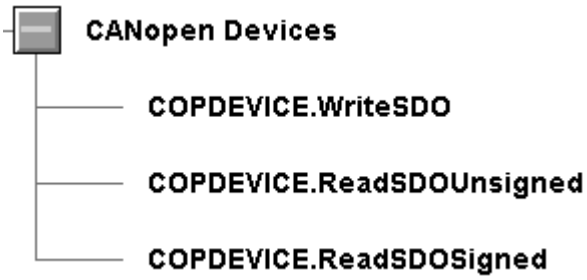
Der CANopen-Baustein verfügt über folgende Methoden:

Methode	Beschreibung
WriteSDO	SDO schreiben
ReadSDOUnsigned	SDO lesen ohne Vorzeichen
ReadSDOSigned	SDO lesen mit Vorzeichen

Tabelle 18.21 Methoden CANopen Baustein

Im FTL-Editor werden die verfügbaren Methoden nach Eingabe des Punktes automatisch in einer Liste angezeigt und können dort ausgewählt werden.

Über das Handbediengerät können diese Methoden über den Menübaum aus der Rubrik „CANopen Geräte“ ausgewählt werden.



Die Abbildung zeigt die Auswahl der Methoden des CANopen-Bausteins am Handbediengerät.

18.10.3 SDO schreiben, Methode WriteSDO

Mit der Methode **WriteSDO** können Datenpakete auf den CAN-Bus geschrieben werden.

Syntax:

‹COPDEVICE›.**WriteSDO** (‹index› : DINT,
‹subindex› : DINT,
‹data› : DINT,
‹type› : SdoDataType) : BOOL

Parameter	Typ	Bedeutung
index	DINT	Index des SDO
subindex	DINT	Subindex des SDO
data	DINT	Daten, die geschrieben werden
type	SdoDataType	Datentyp (Enum) eight_bits (8 Bit) sixteen_bits (16 Bit) thirtytwo_bit (32 Bit)

Tabelle 18.22 Parameter Methode WriteSDO

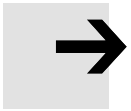
Wird das Paket erfolgreich geschrieben, wird TRUE zurückgegeben, ansonsten FALSE.

Beispiel:

```
oData := 12000
Axis3_copd.WriteSDO(16#6085, 0, oData, thirtytwo_bits)
```

18.10.4 SDO lesen, Methode ReadSDOSigned

Mit der Methode **ReadSDOSigned** können Datenpakete mit Vorzeichen vom CAN-Bus gelesen werden.



Syntax:

‹COPDEVICE›.**ReadSDOSigned** (‹index› : DINT,
‹subindex› : DINT,
‹data› : DINT) : BOOL

Parameter	Typ	Bedeutung
index	DINT	Index des SDO
subindex	DINT	Subindex des SDO
data	DINT	Daten, die gelesen werden

Tabelle 18.23 Parameter Methode ReadSDOSigned

Wird das Paket erfolgreich gelesen, wird TRUE zurückgegeben, ansonsten FALSE.

Beispiel:

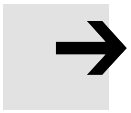
```
dint_iData : DINT := 0
```

Programm:

```
Axis3_copd.ReadSDOSigned(16#6098, 0, dint_iData)
SetInfo("Gelesene Referenzfahrtmethode: %1", dint_iData)
:
```

18.10.5 SDO lesen, Methode ReadSDOUnsigned

Mit der Methode **ReadSDOUnsigned** können Datenpakete auf dem CAN-Bus gelesen werden. Die Methode erkennt automatisch wie lang die vom CAN-Bus gelesenen Daten sind. Diese werden vorzeichenlos als 32 Bit Wert zurückgegeben.



Syntax:

‹COPDEVICE›.**ReadSDOUnigned** (‹index› : DINT,
‹subindex› : DINT,
‹data› : DWORD) : BOOL

Parameter	Typ	Bedeutung
index	DINT	Index des SDO
subindex	DINT	Subindex des SDO
data	DWORD	Daten, die gelesen werden

Tabelle 18.24 Parameter Methode ReadSDOUnsigned

Wird das Paket erfolgreich gelesen, wird TRUE zurückgegeben, ansonsten FALSE.

19. Peripheriesignale

19.1 Arbeiten mit digitalen Ein- und Ausgängen

Digitale Signale werden über die Konfiguration festgelegt. In dieser wird für jedes Hardwaresignal ein eigener Name definiert. Diese Namen können im Bewegungsprogramm für Abfragen oder zum Ansteuern verwendet werden.

Jedes digitale Signal ist in Form eines strukturierten Datentyps abgebildet. Dieser enthält die Daten:

State : BOOL Status des Signals TRUE oder FALSE

Error : BOOL Informationen über einen Fehler des Signals

Der Zugriff auf diese Daten geschieht über den Hardwarenamen und den Punktoperator.

Beispiel:

Ein Sensoreingangssignal hat die Bezeichnung „GripperOpen“. Der Zugriff auf den Zustand des Sensors lautet:

```
GripperOpen.State
```

Dieser Ausdruck ist vom Datentyp BOOL und kann wie eine boolesche Variable im Programm verwendet werden.

19.2 Arbeiten mit analogen Ein- und Ausgängen

Analoge Signale werden über die Konfiguration festgelegt. In dieser wird für jedes Hardware-Signal ein eigener Name definiert. Der Name kann im Bewegungsprogramm wie eine Variable verwendet werden und somit kann z.B. ein Analogwert für eine Berechnung verwendet werden.

Jedes analoge Signal ist in Form eines strukturierten Datentyps abgebildet. Dieser enthält die Daten:

Value : REAL Wert des Signals

Error : BOOL Informationen über einen Fehler des Signals

Der Zugriff auf diese Daten geschieht über den Hardwarenamen und den Punktoperator.

Beispiel:

Ein Analogeingangssignal hat die Bezeichnung „Level“. Der Zugriff auf den Zustand des Sensors lautet:

```
Level.Value
```

Dieser Ausdruck ist vom Datentyp REAL und kann wie eine Variable im Programm verwendet werden.

20. Beispiele

In den folgenden Kapiteln sind Beispiele abgebildet, die die Verwendung der Programmiersprache darstellen.

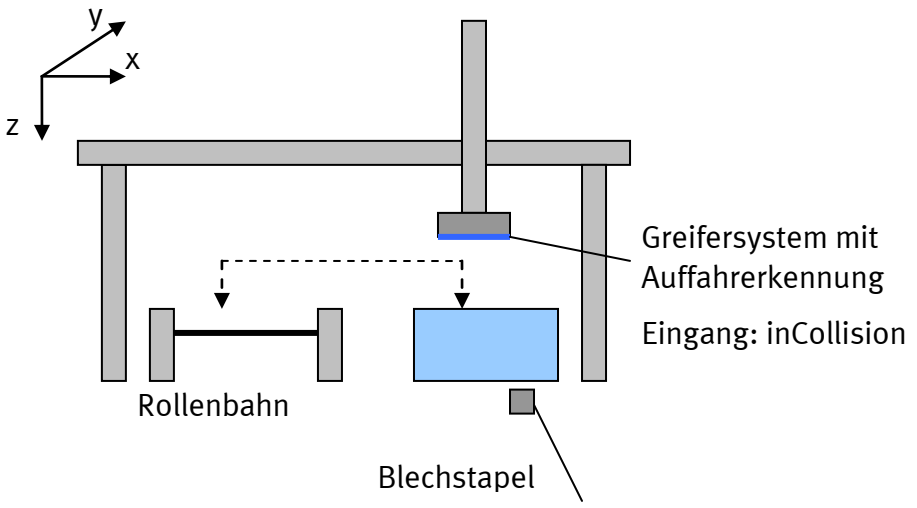
Warnung

Alle Beispiele dienen zur Darstellung der Verwendung der Programmiersprache. Es besteht kein Anspruch auf Richtigkeit, Vollständigkeit und Verwendung der Beispiele als lauffähige Applikation. Werden Programmteile aus den Beispielen in eine Applikation eingebaut, so sind diese bezüglich dem Verwendungszweck zu prüfen und die Funktion mit einer entsprechenden Inbetriebnahme sicher zu stellen.

Die Nichtbeachtung kann Sachschäden und Personenschäden zur Folge haben.

20.1 Stoppen von Bewegungen

Ein Handhabungsroboter muss von einem Stapel Blech eine Tafel entnehmen und auf einer Rollenbahn ablegen. Die genaue Höhe des Stapels ist unbekannt. Das Greifersystem ist jedoch mit einem Sensor ausgestattet, der eine Abfrage erlaubt, ob das Greiferwerkzeug auf ein Hindernis aufgefahren ist. Dieser Sensor wird dazu genutzt, die obere Blechtafel zu detektieren. Um zu erkennen, dass ein Blechstapel abgearbeitet ist, ist zusätzlich ein Sensor (zum Erkennen leerer Stapel) angebracht.

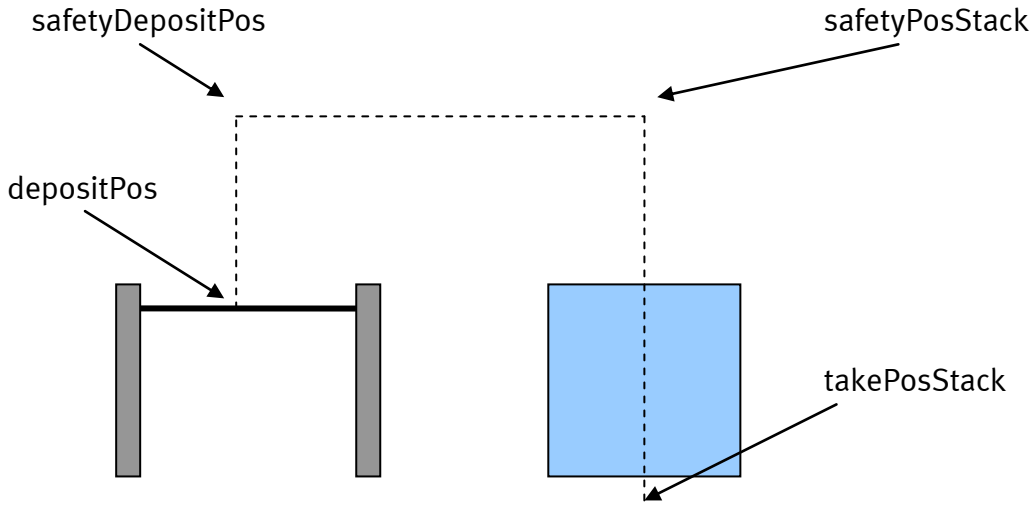


Das Bewegungsprogramm hat folgende Positionen:

Beschreibung	Name der Positionsvariable
Entnahmeposition am Blechstapel	takePosStack
Sicherheitsposition über dem Blechstapel	safetyPosStack
Sicherheitsposition über der Rollenbahn	safetyDepositPos
Ablageposition auf der Rollenbahn	depositPos
Veränderbare Vorposition über Blechstapel	prePosition

Tabelle 20.1 Positionsübersicht

Für diese Anwendung können weitere Positionen eingesetzt werden. Das Beispiel beschränkt sich jedoch auf die angegebenen.



Da Aufgrund der variierenden Stapelhöhe die Höhe der Entnahmeposition unbekannt ist, wird diese unter den Stapel gelegt.

Bewegungsprogramm:

```
:
Vel(dynCart, 200)                // langsame Geschwindigkeit
prePosition := safetyPosStack
WHILE TRUE DO
  Lin(safetyPosStack)
  WAIT inStackNotEmpty.State     // prüfen, ob Teile auf Stapel
  Lin(prePosition)
  Vel(dynCart, 200)              // langsame Geschwindigkeit
  Lin(takePosStack)
  WAIT inCollision.State         // warte Sensorsignal aufgefahren
  StopMove()                     // stoppen der Bewegung
  // ermitteln der Istposition über dem Stapel
  ReadActualPos(prePosition)
  // Sicherheitsabstand über dem Stapel berechnen
  prePosition.z := prePosition.z + 10
  gripper.Set()                  // Greifer schließen
  Vel(dynCart, 1000)             // schnelle Geschwindigkeit
  Lin(safetyPosStack)
  Lin(safetyDepositPos)
  Lin(depositPos)
  gripper.Reset()                // Greifer Öffnen
  Lin(safetyDepositPos)
END_WHILE
:
```

Funktionsweise

Die Kinematik fährt so lange auf die Entnahmeposition, bis sich der Greifer auf der Blechtafel befindet und der Sensor zur Auffahrerkennung das entsprechende Signal liefert. Dies wird mit einer langsamen Geschwindigkeit durchgeführt, um das System vor Schaden zu bewahren.

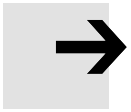
Um im nächsten Zyklus eine möglichst nahe Position über dem Stapel zu erreichen, wird die tatsächliche Position über dem Stapel gespeichert. Diese Position wird mit einem Sicherheitsabstand ergänzt. Somit kann im nächsten Zyklus mit einer schnellen Geschwindigkeit über den Stapel gefahren werden. Anschließend wird bis zum Teilekontakt auf eine langsame Geschwindigkeit umgeschaltet.

20.2 Arbeiten mit Ein- und Ausgangsbausteinen

Bei der Instanzierung von Ein- bzw. Ausgangsbausteinen kann ein Timeout definiert werden. Wird zum Beispiel die Funktion Wait des Bausteins im Ablauf verwendet, muss ein eventuell aufgetretener Timeout vom Anwenderprogramm ausgewertet werden.

Bewegungsprogramm:

```
:
inSensor.Wait()
IF inSensor.Read() = FALSE THEN
   //Ablauf unterbrechen
   SetError("Timeout over")
   RETURN
END_IF
//Ablauf fortsetzten
Lin(pos)
:
```



Hinweis

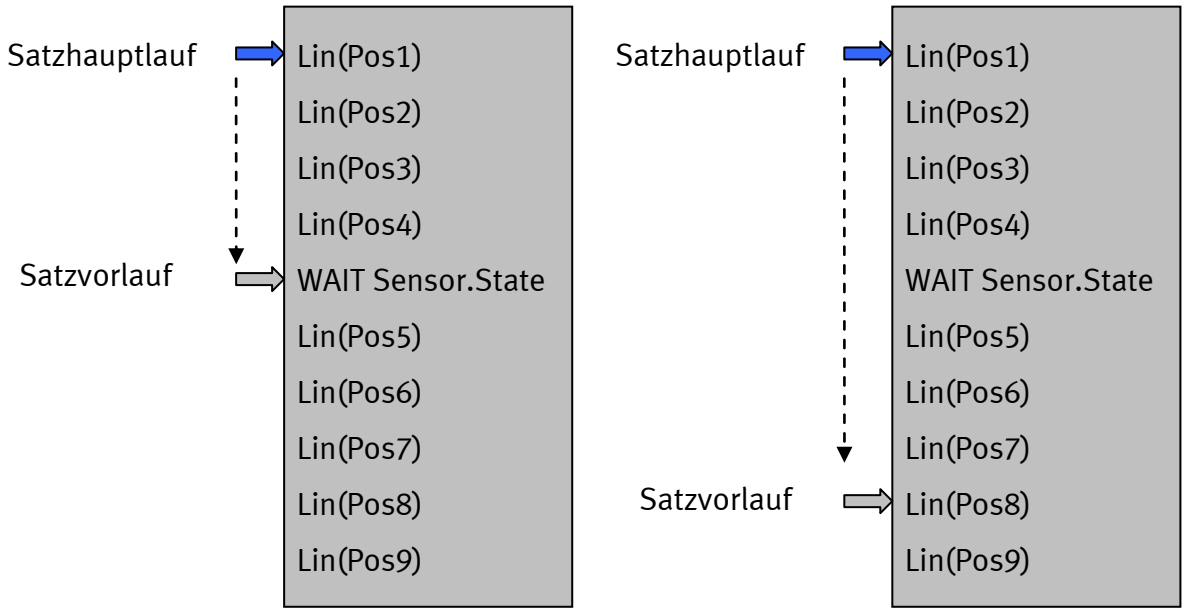
Die Ausgabe einer Fehlermeldung führt zu einer Unterbrechung der Bewegung. Erst nach dem Quittieren der Fehlermeldung kann der Ablauf fortgesetzt werden.

20.3 Steuerung des Satzvorlaufs

Im Folgenden werden anhand von Beispielen verschiedene Möglichkeiten erläutert, wie die Berechnung des Satzvorlaufs beeinflusst werden kann.

1. Warten auf digitales Eingangssignal

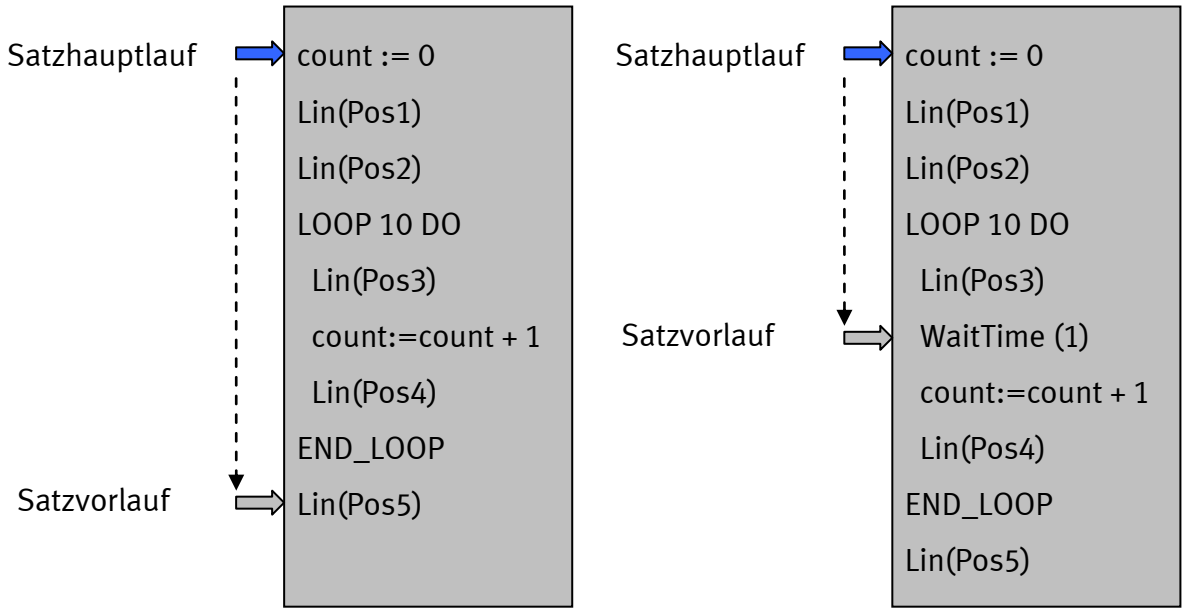
Im Programmablauf wird mit dem Befehl WAIT auf ein digitales Eingangssignal gewartet. Ist der Zustand des digitalen Eingangs FALSE, wird der Satzvorlauf gestoppt. Wird der Zustand TRUE, dann wird die Berechnung fortgesetzt. Dies zeigt die folgende Abbildung.



2. Schleifenzähler

Ein Programmteil wird mit einer Zählschleife 10 mal wiederholt. Die Zyklen werden mit einem Schleifenzähler gezählt. Der Satzvorlauf interpretiert das Ablaufprogramm und erhöht dabei den Schleifenzähler. Dies führt dazu, dass der Satzvorlauf sehr schnell das Programmende erreicht und somit der Schleifenzähler den Wert 10 erreicht. Der Satzhauptlauf hat in dieser Zeit erst die Anweisung Lin(Pos1) erreicht.

Soll der Schleifenzähler den tatsächlichen Zyklus anzeigen, dann muss der Satzvorlauf mit einer WaitTime-Anweisung unterbrochen werden. Der Schleifenzähler wird nun immer beim Erreichen der Pos3 erhöht. Dies zeigt die folgende Abbildung.



20.4 Arbeiten mit Greifern

In der Handhabungstechnik kommen eine Vielzahl verschiedener Greifer zum Einsatz. Jeder Greifvorgang benötigt Zeit, um den Greifer zu schließen bzw. zu öffnen. Diese Zeit kann die Taktzeit wesentlich beeinflussen.

Hinweis

Greifzeiten führen zu einer Erhöhung der Taktzeit und sind deshalb stets zu berücksichtigen. Je nach Greifertyp erfordert dies auch eine Wartezeit im FTL-Programm um das Öffnen oder Schließen des Greifers sicherzustellen.

Hinweis

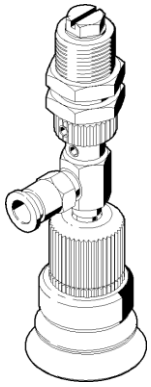
Wartezeiten für den Zustand Geschlossen oder Geöffnet bei pneumatischen Greifern sind immer vom eingestellten Druck abhängig. Ändert sich der Druck, so sind auch die Zeiten anzupassen. Sind am Greifersystem Sensoren für die Endlagen angebracht, so kann in der Regel auf Wartezeiten verzichtet werden.

In den folgenden Kapiteln ist die Integration unterschiedlicher Greifertypen in die CMXR Welt dargestellt.

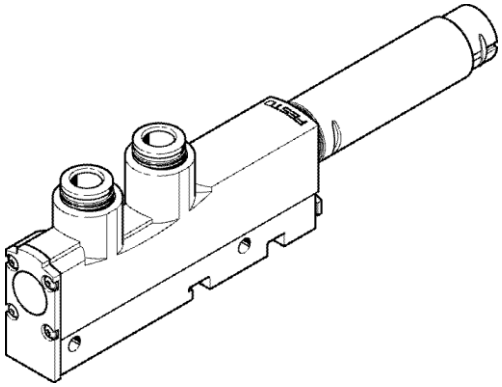
20.4.1 Vakuumgreifer

Vakuumgreifer sind Greifertypen, mit denen sehr schnelle Greifzyklen ausgeführt werden können. Sie benötigen ein Vakuum, dass z. B. durch eine Venturidüse erzeugt werden kann.

Bei der Aufnahme des Werkstückes muss das Vakuum anstehen. Wird das Werkstück abgelegt, muss neben dem Ausschalten des Vakuums ein Abblasimpuls ausgeführt werden, um das im Greifer aufgebaute Vakuum zu beseitigen. Vakuumerzeuger wie der VN-..-H-...-**A** führen nach dem Abschalten des Vakuums den Abblasimpuls automatisch aus. Bei anderen Typen muss dieser Abblasimpuls durch ein zusätzliches digitales Signal angesteuert werden.



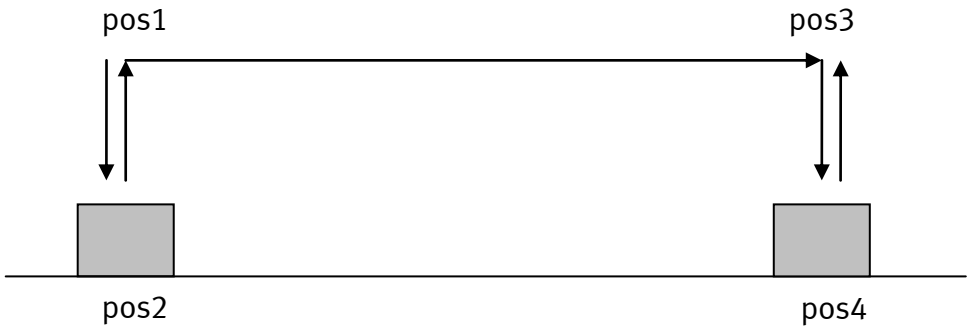
Vakuumgreifer ESG



Vakuumerzeuger VN-..-H-...-A mit eingebautem Ablassimpuls

Beispiel:

Ein Werkstück soll von einer Position pos2 auf die Position pos4 umgesetzt werden.



Als Vakuumerzeuger wird eine Venturidüse eingesetzt. Dieser wird über einen digitalen Ausgang der CMXR angesteuert. Dazu wird ein digitaler Ausgangsbaustein angelegt.

Variablen:

```
vacuum : DOUT := (...)      // digitaler Ausgangsbaustein
```

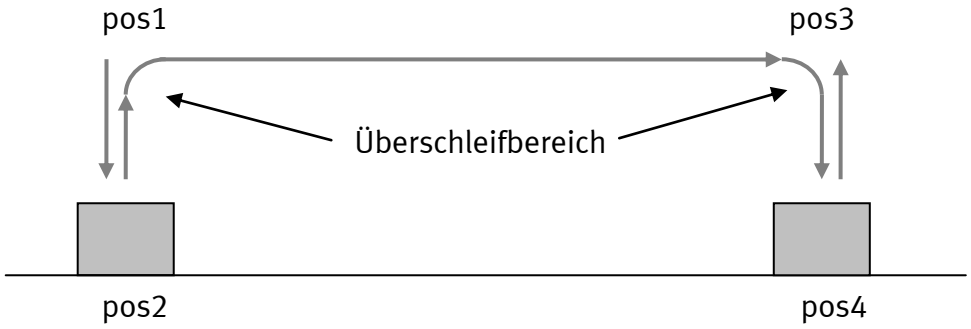
Programm:

```
Vel(dynCart, 1500)   // Bahngeschwindigkeit = 1500 mm/sec
Acc(dynCart, 5000)   // Bahnbeschleunigung = 5000 mm/sec²
OvlVel(75)           // Geschwindigkeitsüberschleifen mit 75%
Lin(pos1)
vacuum.Set()         // Vakuum sofort einschalten
Lin(pos2)            // Aufnahmeposition
WaitTime(10)         // 10 msec Wartezeit fuer Greifzeit
Lin(pos1)
Lin(pos3)            // ueber Ablageposition
Lin(pos4)            // Ablageposition
vacuum.Reset()       // Vakuum am unteren Punkt ausschalten
WaitTime (20)        // 20 msec Wartezeit fuer Greifzeit
Lin(pos3)
```

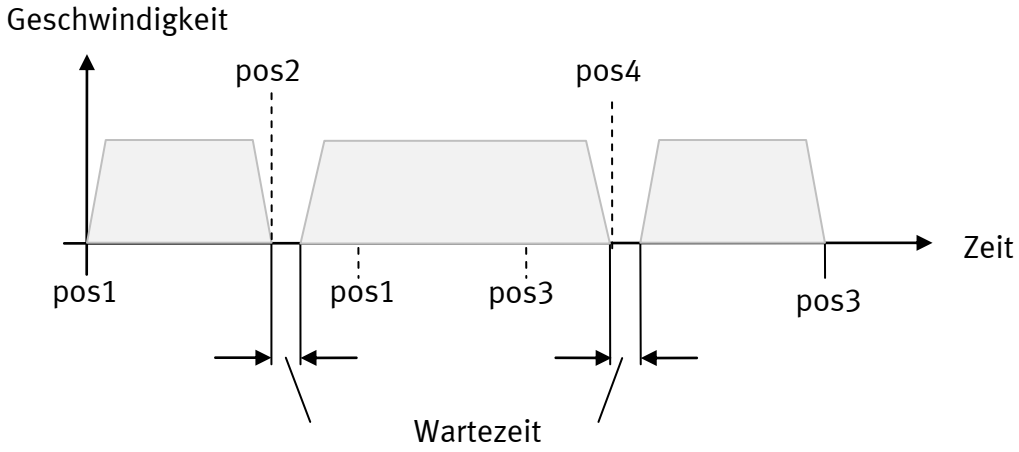
Funktionsweise:

- Da es am Greifer keine Rückmeldungen gibt, ist es erforderlich, beim Schließen und Öffnen eine Wartezeit einzubauen. Diese ist mit Hilfe der Inbetriebnahme zu ermitteln und zu optimieren.
- Die Bewegungen werden mit einem Geschwindigkeitsüberschleifen von 75% abgefahren. Eine Ausnahme bildet hierzu die Aufnahme pos2 und Ablageposition pos3. Der nachfolgende Befehl WaitTime unterbindet ein Überschleifen, da die Satzvorrausberechnung an dieser Stelle angehalten wird und nach Ablauf der Zeit fortgesetzt wird.

Bewegungsprofil:

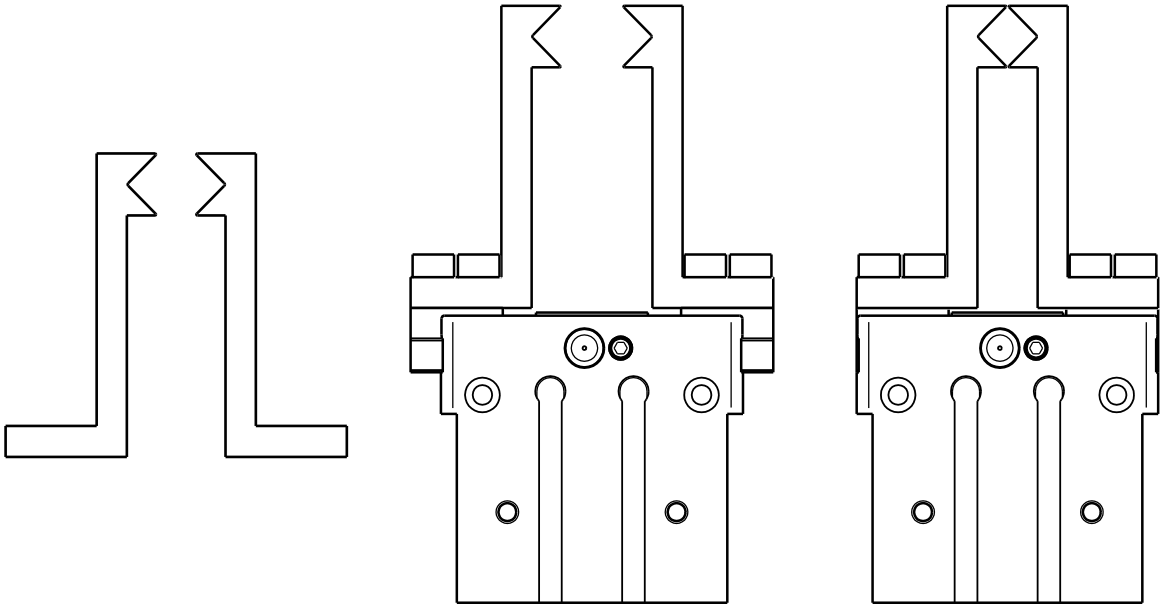


zeitliches Verhalten:



20.4.2 Pneumatische Parallelgreifer

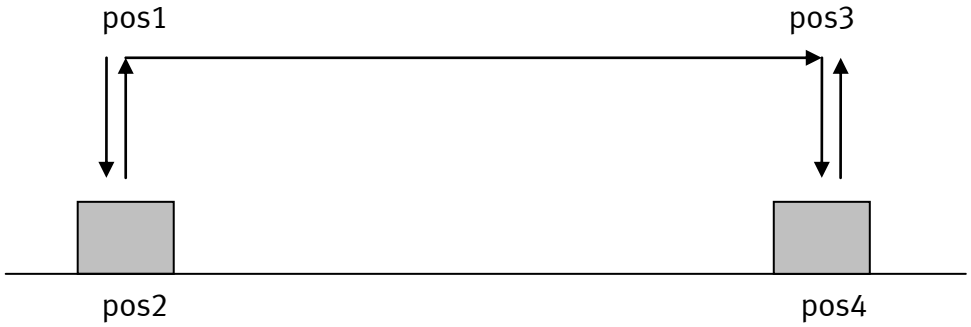
Parallelgreifer besitzen Greiferbacken, die parallel geschlossen werden. Die Greiferbacken können dabei individuell auf die Applikation angepasst werden.



Üblicherweise werden Parallelgreifer zum Öffnen und Schließen mit zwei separaten Ausgängen angesteuert. Optional können an den Greifer Sensoren integriert werden um die Rückmeldungen über die Endlagen (geöffnet oder geschlossen) zu bekommen.

Beispiel:

Ein Werkstück soll von einer Position pos2 auf die Position pos4 umgesetzt werden.



Zum Einsatz kommt ein Parallelgreifer, der über Endlagen-Sensorik verfügt. Dabei ergeben sich folgende digitale Signale, die in Form von Bausteinen in der FTL-Programmierung eingebaut werden.

Variablen:

```
outGripperOpen  : DOUT := (...)  // digitaler Ausgangsbaustein
outGripperClose : DOUT := (...)  // digitaler Ausgangsbaustein
inGripperOpen   : DIN  := (...)  // digitaler Eingangsbaustein
inGripperClosed : DIN  := (...)  // digitaler Eingangsbaustein
```

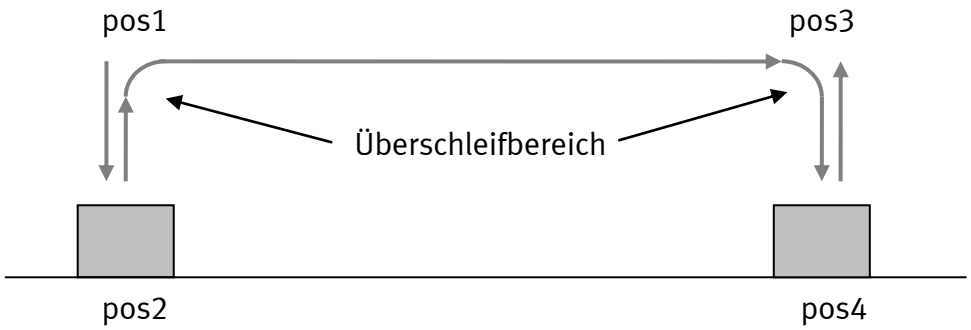
Programm:

```
Vel(dynCart, 1500)  // Bahngeschwindigkeit = 1500 mm/sec
Acc(dynCart, 5000)  // Bahnbeschleunigung = 5000 mm/sec²
OvlVel(75)          // Geschwindigkeitsüberschleifen mit 75%
Lin(pos1)
outGripperOpen.Set()   // Greifer oeffnen
outGripperClose.Reset()
inGripperOpen.Wait(TRUE) // warte bis geoeffnet
Lin(pos2)                // Aufnahmeposition
outGripperClose.Set()    // Greifer schliessen
outGripperOpen.Reset()
inGripperClosed.Wait()    // warte bis geschlossen
Lin(pos1)
Lin(pos3)
Lin(pos4)               // Ablageposition
outGripperOpen.Set()    // Greifer oeffnen
outGripperClose.Reset()
inGripperOpen.Wait()    // warte bis geoeffnet
Lin(pos3)
```

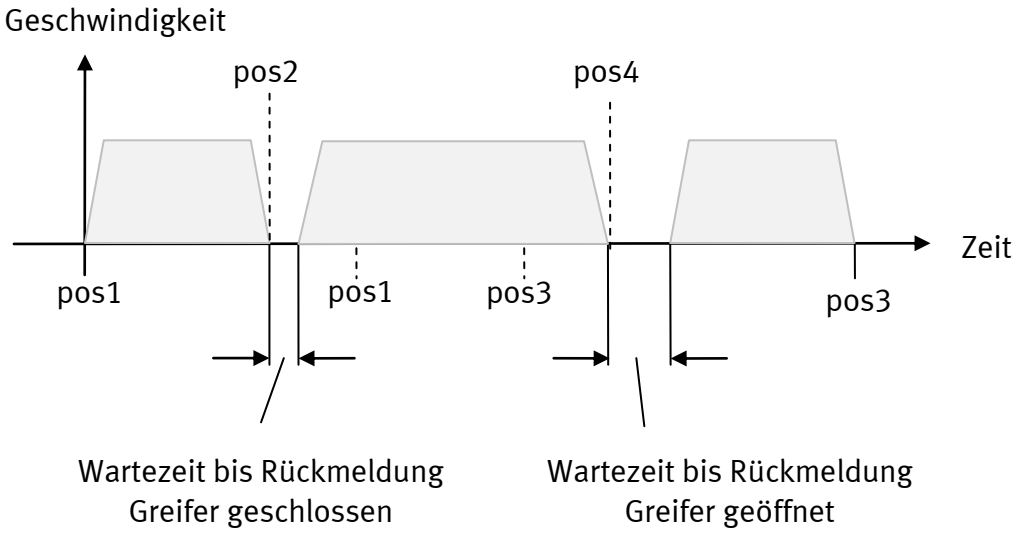
Funktionsweise:

- Bei vorhandenen Sensoren für die Rückmeldung der Greiferendlagen sind keine Wartezeiten nötigt.
- Die Bewegungen werden mit einem Geschwindigkeitsüberschleifen von 75% abgefahren. Durch das Warten auf die Endlagenrückmeldungen mit der Bausteinmethode Wait, wird ein Überschleifen an den Aufnahme- und Ablagepositionen unterbunden. Die Satzvorausberechnung wird an dieser Stelle angehalten und nach Schalten des digitalen Eingangssignales fortgesetzt.

Bewegungsprofil:

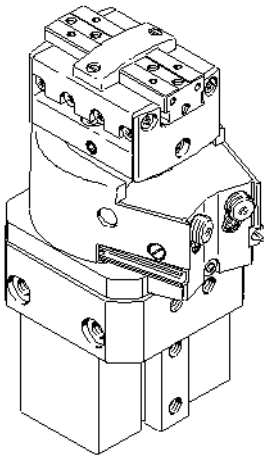


zeitliches Verhalten:



20.4.3 Pneumatische Schwenk-Greifeinheit

Die pneumatische Schwenk-Greifeinheit HGDS von Festo vereint einen Parallelgreifer mit einem Schwenkantrieb zu einer Einheit.

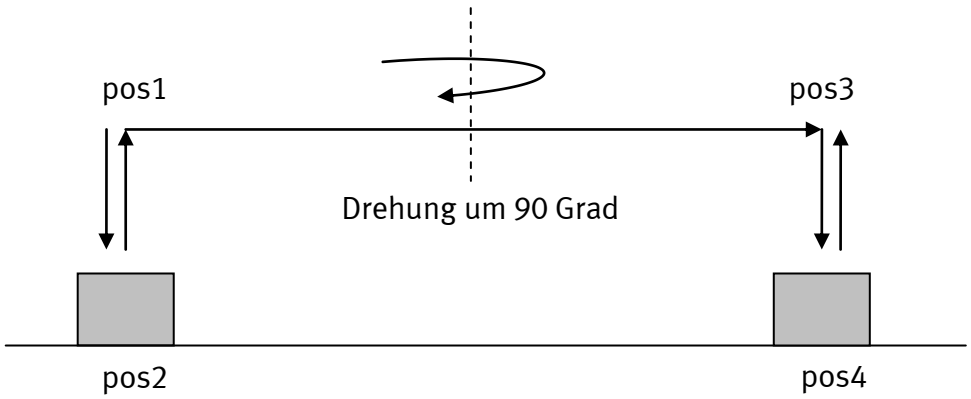


Schwenk-Greifeinheit HGDS
mit Parallelgreifer

Die HGDS bietet die Möglichkeit, Sensoren für die Erkennung der Endlagen des Greifers und des Schwenkantriebs einzubauen. Die Sensoren werden benötigt um die Applikation sicher zu steuern.

Beispiel:

Ein Werkstück soll von einer Position pos2 auf die Position pos4 mit einer Drehung von 90 Grad umgesetzt werden.



Zum Einsatz kommt die HGDS Einheit, die über die entsprechenden Sensoren verfügt, um die Endlagen zu erkennen. Dabei ergeben sich folgende digitale Signale, die in Form von Bausteinen in der FTL-Programmierung eingebaut werden.

Variablen:

```
outGripperOpen  : DOUT := (...)  // digitaler Ausgangsbaustein
outGripperClose : DOUT := (...)  // digitaler Ausgangsbaustein
outPos0Degree   : DOUT := (...)  // digitaler Ausgangsbaustein
outPos90Degree  : DOUT := (...)  // digitaler Ausgangsbaustein
inGripperOpen   : DIN := (...)   // digitaler Eingangsbaustein
inGripperClose  : DIN := (...)   // digitaler Eingangsbaustein
inPos0Degree    : DIN := (...)   // digitaler Eingangsbaustein
inPos90Degree   : DIN := (...)   // digitaler Eingangsbaustein
```

Programm:

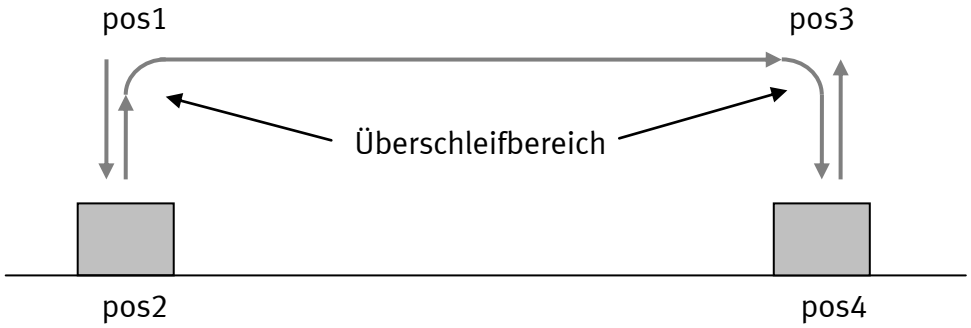
```
Vel(dynCart, 1500)     // Bahngeschwindigkeit = 1500 mm/sec
Acc(dynCart, 5000)     // Bahnbeschleunigung = 5000 mm/sec²
OvlVel(75)             // Geschwindigkeitsüberschleifen mit 75%
Lin(pos1)                // fahre über Teil
outGripperClose.Reset() // Greifer auf
outGripperOpen.Set()
outPos0Degree.Set()      // auf 0 Grad drehen
outPos90Degree.Reset()
inGripperOpen.Wait()     // warten bis Greifer offen
inPos0Degree.Wait()      // warten bis auf 0 Grad
Lin(pos2)                // Aufnahmeposition
outGripperClose.Set()    // Greifer schließen
outGripperOpen.Reset()
inGripperClose.Wait()    // warten bis Greifer geschlossen
Lin(pos1)
outPos0Degree.Reset()    // auf 90 Grad drehen
outPos90Degree.Set()
Lin(pos3)
inPos90Degree.Wait(TRUE) // warten 90 Grad mit Ueberschleifen
Lin(pos4)                 // Ablageposition
outGripperClose.Reset()  // Greifer auf
outGripperOpen.Set()
inGripperOpen.Wait()     // warten bis Greifer offen
Lin(pos3)
```

Funktionsweise

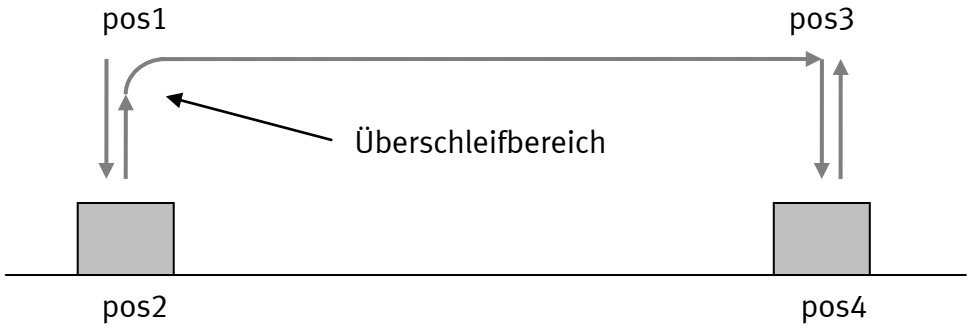
- Mit integrierten Sensoren sind für die Rückmeldungen der Greif- und Drehbewegungen keine Wartezeiten notwendig.
- Die Abfrage der Drehbewegung auf 90 Grad wird mit der Möglichkeit des Überschleifens ausgeführt. Dies bedeutet, wenn die Rückmeldung der Drehung auf 90 Grad beim Fahren auf pos4 nicht vorhanden ist, wird die Bewegung gestoppt und gewartet bis die Rückmeldung ansteht. Liegt die Rückmeldung bereits an, so wird direkt eine Überschleifbewegung auf pos4 ausgeführt.

Bewegungsprofil:

Die Rückmeldung der Drehbewegung auf 90 Grad wird vor dem Erreichen der Position pos3 erkannt. Es wird eine Überschleifbewegung auf pos4 ausgeführt.

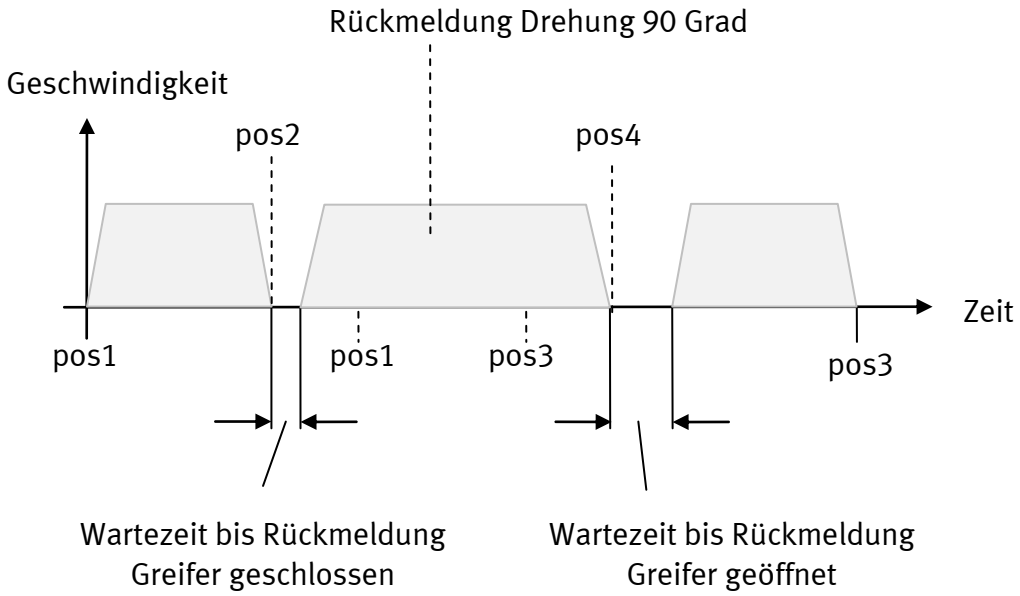


Die Rückmeldung der Drehbewegung auf 90 Grad ist vor dem Erreichen der Position pos3 nicht eingetroffen. Es wird an der Position pos3 auf die Rückmeldung gewertet und keine Überschleifbewegung auf pos4 ausgeführt.

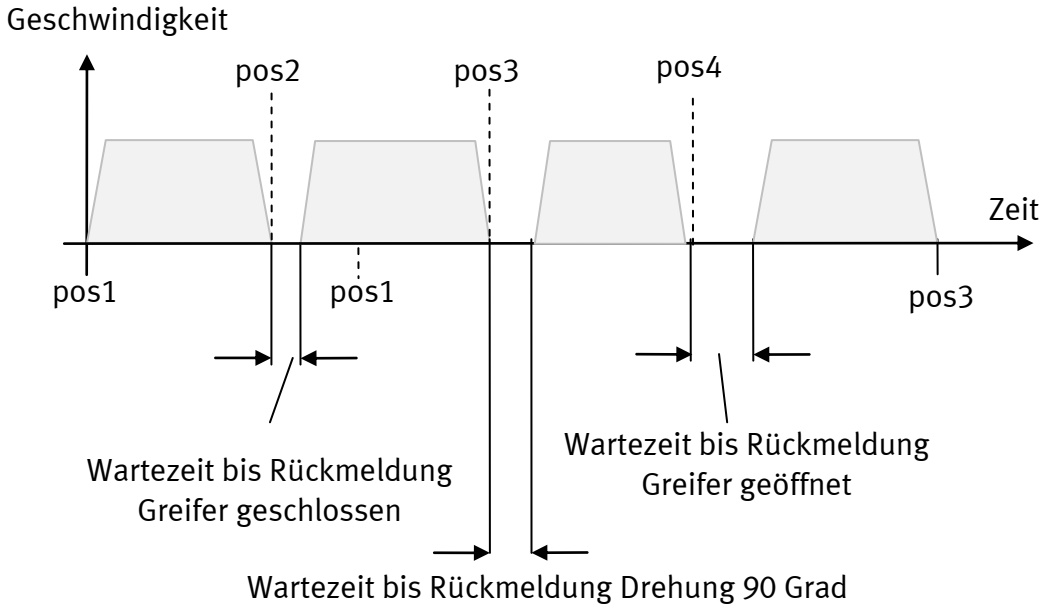


Zeitliches Verhalten:

Rückmeldung der Drehung auf 90 Grad ist vor dem Erreichen der Position pos3 vorhanden. Es wird eine Überschleifbewegung auf Position pos4 ausgeführt.



Rückmeldung der Drehung auf 90 Grad ist nicht an Position pos3 vorhanden. Es wird auf diese gewartet.



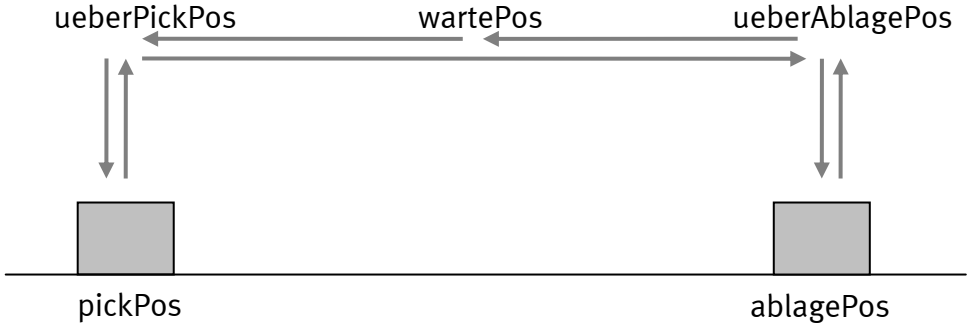
20.5 Arbeiten mit der SPS-Schnittstelle

20.5.1 Aufgabenstellung

Im folgenden Beispiel werden Teile von der Position „pickPos“ abgeholt und auf die Position „ablagePos“ transportiert. Dabei ist die Abholposition dynamisch. Die genauen Koordinaten werden in jedem Zyklus von einem Kamerasystem berechnet und über die PROFIBUS-SPS-Schnittstelle an die Steuerung gesendet.

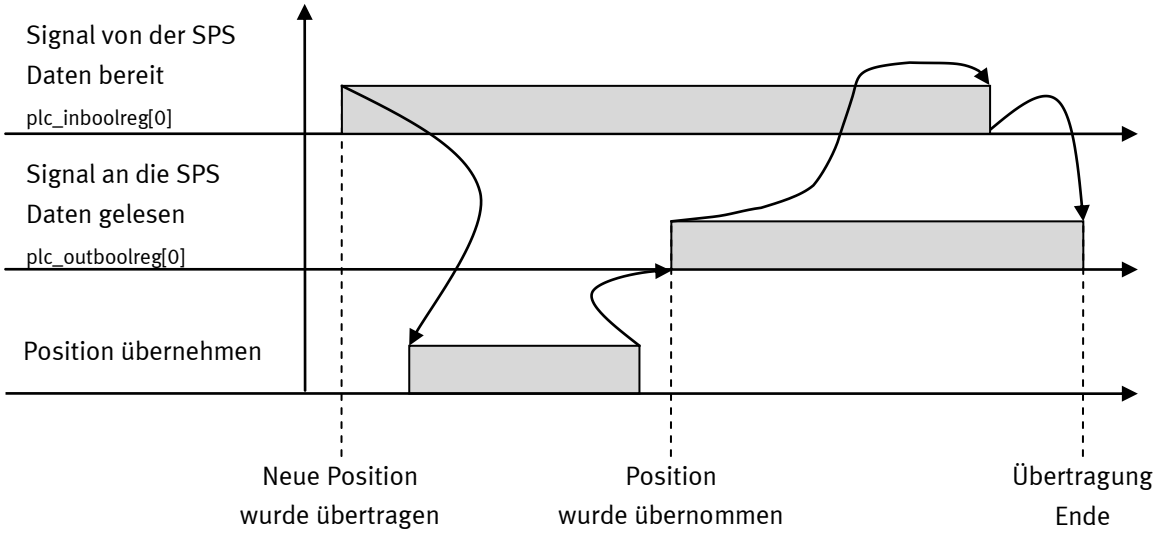
Für den Ablauf werden aus der Start- und Endposition zwei dynamische Zwischenpositionen berechnet. Diese werden im Ablauf als Stützpunkte benutzt.

Werden von der SPS nicht rechtzeitig Daten geliefert, so wird der Zyklus auf der Position „wartePos“ angehalten, bis die SPS bereit ist.



20.5.2 SPS-Schnittstelle

Der Datenaustausch zwischen der CMXR Steuerung und der SPS wird mit zwei Synchronisierungsbits realisiert. Im ersten Schritt wird von der SPS signalisiert, dass Daten gesendet wurden. Sobald die Daten von der CMXR übernommen wurden, wird dies an die SPS gemeldet. Anschließend werden beide Schnittstellensignale zurückgesetzt. Der Ablauf ist im folgenden Ablaufdiagramm dargestellt.



20.5.3 Ablaufprogramm

Variablen:

```
waitPos          : CARTPOS := (...)
pickPos          : CARTPOS := (...)
depositPos       : CARTPOS := (...)
abovePickPos     : CARTPOS := (...)
aboveDepositPos  : CARTPOS := (...)
gripper          : DOUT := (...)
```

Programm:

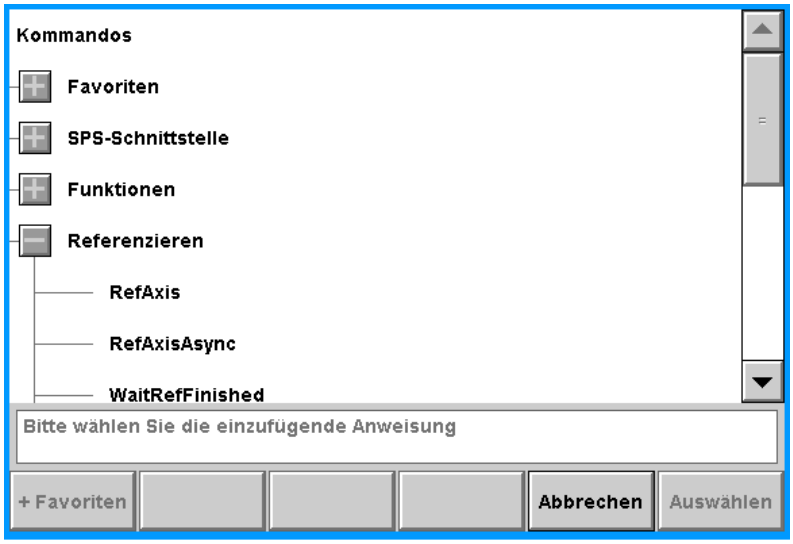
```
// Initialisierung
Acc(dynCart, 4000)
Vel(dynCart, 100)

Lin(waitPos)

WHILE TRUE DO
  // Handshake
  WAIT plc_inboolreg[0] // SPS signalisiert eine neue Position
  pickPos := plc_cposreg[0]
  abovePickPos.x := plc_cposreg[0].x
  abovePickPos.y := plc_cposreg[0].y
  plc_outboolreg[0] := TRUE // Bestätigt die Übernahme
  WAIT NOT plc_inboolreg[0] // Warten auf Reaktion der SPS
  plc_outboolreg[0] := FALSE
  // Anfahren der Aufnahmeposition
  Lin(abovePickPos)
  Lin(pickPos)
  // Greifen
  gripper.Set()
  WaitTime(200)
  // Anfahren der Ablageposition
  Lin(abovePickPos)
  Lin(aboveDepositPos)
```

```
  Lin(depositPos)
  // Ablegen
  gripper.Reset()
  WaitTime(200)
  // Anfahren der Warteposition
  Lin(aboveDepositPos)
  Lin(waitPos)
END_WHILE
```

21. Menübaum der FTL Befehle

Um die Programmierung zu erleichtern, wird am Handbediengerät CDSA und im FTL Programmiereditor ein Menübaum angezeigt, der alle verfügbaren FTL Befehle enthält. Im folgenden werden alle Befehle des Befehlsumfangs „FTL Programmierung Basis“ dargestellt.

Darstellung des Menübaums am Handbediengerät CDSA:



SPS-Schnittstelle

- ProgHold

Funktionen

Positionsfunktionen

- ReadTargetPos
- ReadActualPos
- SavePosition

- SIN
- COS
- TAN
- COT
- LN
- ABS
- SQRT
- EXP
- ASIN

- ACOS
- ATAN
- ATAN2
- ACOT
- Time
- TimeToStr

Referenzieren

- RefAxis
- RefAxisAsync
- WaitRefFinished
- IsAxisReferenced

Bewegungsbefehle

- Ptp
- Lin
- CircIp
- LinToCircIp
- PtpToCircIp
- MoveAxisPtp
- MoveAxisCart
- PtpRel
- LinRel
- StopMove

Dynamikbefehle

- DynOvr
- Vel
- Acc
- Jerk
- Ovr
- OvlVel
- OvlCart
- Ramp
- VconstOn
- VconstOff

Anweisungen

- WaitOnPath
- StopProgram
- WaitTime
- WaitOnPos
- CALL
- RUN
- KILL
- ... := ...
- //
- WAIT
- LABEL
- GOTO
- IF ... GOTO
- IF ... THEN ... END_IF
- ELSE
- ELSIF ... THEN
- WHILE ... DO ... END_WHILE
- LOOP
- RETURN

Roboterbefehle

Referenzsysteme

- SetRefSys
- SetRefSys3P
- SetRefSysWorld

- Tool

Systemfunktionen

Zeitmessung

- CLOCK.Reset
- CLOCK.Start
- CLOCK.Stop
- CLOCK.Read
- CLOCK.ToStr

zusätzliche Funktionen

- SHR

- SHL
- ROR
- ROL
- STR
- CHR
- ORD

Meldesystem

- SetInfo
- SetWarning
- SetError

E/A Bausteine

Analoge Eingänge

- AIN.Read
- AIN.WaitLss
- AIN.WaitGrt
- AIN.WaitIns
- AIN.WaitOuts

Analoge Ausgänge

- AOUT.Write
- AOUT.Read
- AOUT.WaitLss
- AOUT.WaitGrt
- AOUT.WaitIns
- AOUT.WaitOuts

CANopen Geräte

- COPDEVICE.WriteSDO
- COPDEVICE.ReadSDOUnsigned
- COPDEVICE.ReadSDOSigned

Digitale Eingänge

- DIN.Read
- DIN.Wait
- DIN.WaitN
- DIN.RisingEdge
- DIN.ResetRisingEdge

- **Digitale Ausgänge**
  - DOUT.Set
  - DOUT.Reset
  - DOUT.Write
  - DOUT.Pulse
  - DOUT.Read
  - DOUT.ResetRisingEdge
  - DOUT.Wait
  - DOUT.WaitN
  - DOUT.RisingEdge
- **Inkrementalgeber**
  - ENCODER.Set
  - ENCODER.Read

#

##....61

/

//....61

_

_global.tt....19

A

ABS....152
Acc....91
Achsposition....63
ACOS....147
ACOT....150
AIN....170
AOUT....175
Array....31
ASIN....146
ATAN....149
ATAN2....151
AXISPOS....63

B

Beispiel
- Arbeiten mit der SPS Schnittstelle ..210
- Arbeiten mit Ein- und Ausgangsbausteinen....197
- Arbeiten mit Greifern....199
- Steuerung des Satzvorlauf....198
- Stoppen von Bewegungen....194
- WAIT mit digitalem Eingang....57

C

CALL....51
CARTPOS....63
CHR....155
CircIp....81
CLOCK....180
COS....147
COT....150

D

Datentypen....28
deaktivierter Programmsatz....61
DIN....159
DO....39
DOUT....163
DynOvr....96

E

ENCODER....184
Error....192
EXP....151

F

FTL....12, 16, 19

G

GOTO....48

I

IF...GOTO....47
IF...THEN....45
IS_MAPPED....37
IsReferenced....121

J

Jerk....92

K

Kartesische Position....66
KILL....54
Kommentar....61

L

LABEL....47
Lin....74
LinRel....76
LinToCircIp....85
LN....151
LOOP....49
LOW, HIGH....34

M

MAP....37
MAPTO....37
MoveAxisCart....72
MoveAxisPtp....72

N

Nullsegment....103

O

ORD....155
OvlCart....107
OvlVel....105

Ovr 94

P

plc_AxisPos 133
plc_CartPos 133
plc_RefSys 134
plc_VarDint 132
Profibusschnittstelle 130
ProgHold 135
Ptp 67
PtpRel 70
PtpToCircIp 83

R

Ramp 98
Read 187
ReadActualPos 142
ReadSDOSigned 189
ReadSDOUnsigned 191
ReadTargetPos 143
RefAxis 116
RefAxisAsync 118
Referenzsysteme 134
RETURN 52
ROL 154
ROR 154
RUN 54

S

Satzvorlauf 54, 56, 198
SavePosition 144
Schlüsselwörter 24
Set 186
SetError 141
SetInfo 139
SetRefSys 111
SetRefSys3P 112
SetRefSysWorld 114
SetWarning 140
SHL 153
SHR 153
SIN 146
SQRT 152
State 192
StopMove 86
StopProg 87
STR 155

T

TAN 149
TCP 122
tid 19, 20
Time 144
TimeToStr 145
tip 19
Tool 125
Typumwandlungen 30

V

Value 192
VconstOff 101
VconstOn 99
Vel 90

W

WAIT 56
WaitOnPath 58
WaitOnPos 60
WaitRefFinished 120
WaitTime 54
Wartezeit 54
Werkzeuge 122
WHILE 48
WriteSDO 189